CREATING ARCADE GAMES ON THE TI-99/4A

A step-by-step guide to writing your own games on the TI-99/4A®, including eight complete games ready to type in and play.

Seth McEvoy

A **COMPUTE! Books** Publication

$12.95

CREATING ARCADE GAMES ON THE TI-99/4A

Seth McEvoy

COMPUTE! Publications, Inc.
One of the ABC Publishing Companies
Greensboro, North Carolina

Printed in the United States of America
ISBN 0-942386-27-2

10 9 8 7 6 5 4 3 2 1

COMPUTE! Publications, Inc., Post Office Box 5406, Greensboro, NC 27403 (919) 275-9809, is one of the ABC publishing companies, and is not associated with any manufacturer of personal computers. TI-99/4A is a trademark of Texas Instruments Inc.

Contents

Foreword

Whether you're just beginning to program or have been computing for years, *Creating Arcade Games on the TI-99/4A* will show you how to write an arcade-style game. Step by step, this book will show you how to create game characters, move them on the screen, control them with the keyboard or joystick, produce sound effects, and even keep score.

How do you create a videogame? How can you make figures move about on a colorful screen? How can you make the computer control some of the figures while you control others by pressing keys or moving the joystick?

When you play arcade-style games, you take these things for granted. But when you first try to create your own videogame, you may find them difficult problems to solve.

You'll use the TI-99/4A's built-in graphic and sound abilities to enhance your games, and even see how to draw and move sprites with the TI Extended BASIC cartridge. Starting with an idea, you'll see how to develop a complete game by following a simple step-by-step guide, from choosing the game's concept to testing and debugging the finished program.

Also included are eight complete games ready to type in and enjoy. Because they're fully explained, these games will quickly lead you to a full understanding of the techniques necessary to create your own exciting games. There's even a game written with TI Extended BASIC that uses sprites to create an action-packed game of speed and skill.

Like all COMPUTE! books, you'll refer to this book not just once, as you write your first game, but again and again. You'll find it a valuable resource for more complex game ideas and information on the TI-99/4A as you design other videogames. With the ideas in this book, and your creativity, you'll quickly be writing and playing your own arcade-style games.

Acknowledgments

Creating Arcade Games on the TI-99/4A is dedicated to Laure Smith, who made the ideas in this book easier to understand.

1 Starting Out

1 Starting Out

This book is designed to help you create fast-action, arcade-style games, no matter how little or how much you already know about the TI-99/4A computer. If you are new to computers and computer programming, Chapters 2 through 6 will give a clear and simple step-by-step introduction to BASIC, the language that provides the building blocks you'll use to create videogame programs.

All videogames are made up of a small number of programming pieces, called commands, functions, and operations. These pieces are then combined in many different ways to create all kinds of arcade games.

The rest of the book consists of actual game programs, which you can type in and play. Most important is the fact that you can also *change* these programs. Experimenting with the programs will help you because you'll often learn more from seeing what changes when you alter a working program than you can from discussions or examples of theory. By experimenting, you can discover a new and better way to do something. Above all, don't worry about damaging your computer by experimenting with the programs. *Nothing you type in can possibly hurt your TI-99/4A.*

Even though BASIC is one of the easier computer languages to learn, it is still a language, like French or Spanish. You must learn it step by step, one word at a time. Some parts of the language will be easy for you to learn and understand, and other parts may seem more difficult. Reread the difficult parts and be patient. It takes time to make a new language natural and comfortable.

If you already know TI BASIC, you can skim the early chapters until you find material new to you. However, if you know BASIC only from other computers, it would be a good idea to read everything. The TI-99/4A has features that you'll need to be aware of.

Chapters 7 through 13 each contain a videogame, written to show you how the TI-99/4A computer can be used to create arcade-style games. You'll be shown how you can modify the game to make it harder or easier, faster or slower. At least one of these modifications will be outlined, complete with the program lines you'll need to alter or add. You'll also be given hints on what you can do to expand the game and make it

more interesting.

Each of the videogames is explained in great detail, and any unusual techniques are explained fully. By the time you have gone through each game, you'll have an arsenal of techniques and tricks that you can use to create your own videogames on the TI computer.

TI has two BASIC languages. The first comes with your computer, and is called TI BASIC. You can purchase a separate cartridge to plug into your computer called TI Extended BASIC. This language lets you use more BASIC keywords, and includes sprites, which are moving characters that make creating games even easier. Chapter 13 has a game that shows you how sprites and Extended BASIC work.

Finally, Chapter 14 is a detailed, step-by-step outline that will show you how to create new and original games. You'll see how to get ideas, how to work those ideas into games, and the steps necessary to make your game programs come alive. By following this simple step-by-step process, and by using the information throughout the book, you should have no trouble designing the games that you've always wanted to play.

Getting Help

Appendix C, "In Case of Error," will help you in your own programming when you make errors. All programmers make errors. The secret is knowing what to do when something goes wrong.

Equipment

All you need to begin programming videogames is your TI-99/4A computer. If you have a TI-99/4 computer, you may be able to adapt these programs to that earlier version of the 99/4A. Most of the differences will be slight; check your TI-99/4 manual to see if the commands work the same. You will, of course, need a TV set, and you should have a tape recorder or disk drive to save your games once you've typed them in.

2 Putting Characters on the Screen

2 Putting Characters on the Screen

The first step in creating your own fast-action arcade game is to learn how to get your TI computer to do exactly what you want it to do.

You give the computer instructions by using keywords that are part of the BASIC language. You'll learn many different BASIC keywords in this book, but the ones that are most important are those that put pictures on the screen.

PRINT is the first BASIC keyword you'll use to put letters, words, and pictures on the screen. These drawings can look like animals, people, spaceships, or *whatever you want.*

Often called *characters,* these pictures can be made to do things and move around. If a game doesn't have moving characters, it isn't a real arcade game.

How to Put Things on the Screen

When you turn on your TI-99/4A, you'll see the beginning rainbow screen. Your TI is ready. Press any key to begin. You should see a message telling you to press *1* for TI BASIC, or 2 for any other cartridges you have in the machine. For now, if you have any other cartridges in your computer, take them out. (But only after turning off the computer—*never* insert or remove a cartridge while the computer is on, because you can damage it.)

Press the *1* key, and the screen will show TI BASIC READY near the bottom. Below it, you'll see an arrow, which tells you that the computer is ready for instructions.

Type PRINT. Nothing happened! This is because you must tell the computer *what* you want to PRINT.

After the word PRINT, add a quote. On the TI-99/4A, you get a quote to appear on the screen by pressing the FCTN key at the lower right of the keyboard and the *P* key at the same time. The FCTN (FUNCTION) key has a gray dot on it and lets you use a key for more than one thing.

After you type the quote, type the letter *A*. Then add

another quote.

Your line should look like this:

PRINT "A"

Nothing has happened yet. To tell the computer that you are finished with that line, press the ENTER key. It has a yellow dot on it and is on the right side of the keyboard.

When you pressed ENTER, several things happened. First of all, the line you typed moved up a few lines, and below it appeared the letter A, all by itself. Then, a new arrow appeared below, telling you that the computer is ready to do something else.

Try using PRINT to put other letters on the screen. For example, you can type:

PRINT "THE TI IS GREAT"

When you press ENTER, the computer will display *THE TI IS GREAT* on the screen.

Note: When you start out with TI BASIC, smaller-sized letters will appear on the screen. You will need to depress the ALPHA LOCK key to get larger letters if you want to type words inside quotes. Whatever is typed inside a pair of quotes will stay that way. Whatever is typed outside quotes, like the word PRINT, will be changed to large letters automatically by the computer.

Making a Program

Line Numbers. You can't make much of a game by just typing PRINT. You must create a program, which allows you to tell the computer to execute more than one line. If you just use PRINT by itself, you'll be limited to seeing only what you have typed. That isn't much fun, and is certainly not a game.

A program is a group of commands and functions that execute in order. You can tell the computer in what order you want it to do things by putting *line numbers* before you type the commands.

For instance, you can type:

```
10 PRINT "TEXAS INSTRUMENTS"
```

This time, when you press the ENTER key, the line moves up, but it doesn't PRINT. This is because the TI is waiting for more numbered lines, or for you to tell it what to do with the collection of numbered lines you have typed. When you typed

Putting Characters on the Screen 2

PRINT "A" by itself, the computer did what you told it to do right away. This is called the *command mode*, and you won't see it used much in game programming, because a program that creates a game must do many things to make a game play.

Line-Numbered Commands. By adding a line number before a command, you're telling the computer to store the line you type, after you press ENTER. The computer will use the line number to keep track of every line you type.

The TI has a special command called NUM which will automatically number the lines for you. You can type NUM and press ENTER, and the lines will start at 100 and increment by 10. If you don't want your line numbers to start at 100 and count up by 10's, you can change this by typing:

NUM *first line, distance between lines*

and pressing ENTER.

For example, if you want to start at line 10, and go up by 5 each time, just type NUM 10,5 and press ENTER.

RUN. When you're ready to use all the commands, you simply type RUN and press ENTER. The computer will start at the line with the lowest number and execute it. After it's finished with that line, it will go on to the line with the next highest number, and so on. It will go through all the line-numbered commands you've stored in memory until it comes to the end. When it's through, it will display:

** DONE **

If it doesn't make it through, you'll see an error message. Refer to Appendix C, "In Case of Error," if you need help.

Running a Program. You have already typed:

```
10 PRINT "TEXAS INSTRUMENTS"
```

into your computer. You can add two more lines as follows:

```
20 PRINT "MAKES TERRIFIC"
30 PRINT "COMPUTERS"
```

When you have finished typing, you can PRINT out all three lines by typing RUN and pressing the ENTER key.

You should see:

TEXAS INSTRUMENTS

MAKES TERRIFIC

COMPUTERS

on the screen.

2 Putting Characters on the Screen

LIST and EDIT. LIST and EDIT are two BASIC keywords that will help you make changes in your line-numbered commands without retyping an entire line.

You may need to see the numbered lines you have stored in memory. You can do this by typing the word LIST and pressing ENTER. You can LIST one line by typing LIST and the line number. For example, typing LIST 10 will print out line 10. You can see a group of lines by typing:

LIST *first line-last line*

and pressing ENTER.

For example, if you want to see lines 10 through 20, type LIST 10-20. You get the hyphen by typing SHIFT/.

When you're writing your programs and make a mistake or want to change a line-numbered statement in your program, you can type EDIT and the number of the line you want to change. That line will appear at the bottom of the screen and anything else on the screen will scroll up one line.

You'll see the cursor after the line number of the line you now want to edit. If the blinking cursor is on top of a letter on the line you're editing, whatever you type will replace what was there. You can use the arrow keys (FCTN and S to move left, FCTN and D to move right) to move the cursor.

If you want to insert a letter or letters, type FCTN and 2. Whatever you type will be inserted *before* the cursor. You can type FCTN and 1 to delete anything under the cursor.

When you're through, press ENTER. The new line will replace the old line. After editing, it's always a good idea to use the LIST command to check your newly edited line.

Another way to edit lines is to enter the line number and then press the FCTN key and the down-arrow key (the *x* key). The line will display and can be edited just as you did using the EDIT command. This method may be more useful if you're editing several lines in sequence. To display the next line, just press the FCTN key and down-arrow key, and it will appear.

Be Careful. When you're writing programs on the TI computer, be careful not to accidentally press the FCTN key and the = key at the same time. *If you do, your program will be erased!* To erase a program, just turn off your TI or type NEW and press ENTER. Typing FCTN and = together can be disastrous, especially if you thought you were typing a plus sign, which is SHIFT and =.

Also to be safe, save your programs frequently to tape or disk so that if there is a power failure, your program won't be lost.

Where Can You Print?

So far you have had no choice as to where PRINT put the letters and words on the screen. When you create your arcade games, however, you'll want to position all elements of your game exactly where you want them on the screen.

The TI screen can be used like a map, and you can use the CALL HCHAR command to put your characters *exactly* where you want them. The TI has several commands that use the word CALL followed by another word. By using CALL, the TI computer knows the word that follows will tell it to do something unusual, something that is not part of the standard BASIC list of keywords used on other computers. CALL HCHAR means to use the word HCHAR, which stands for *H*orizontal *CHAR*acters.

The map is blank when you start, and is divided up into a grid. There are 32 columns across and 24 rows down, making up 768 blocks. Each column and row has a number. The columns are numbered from 1 to 32, with the first column on the left. The rows are numbered from 1 to 24, with the first row at the top. Figure 2-1, TI Screen Map, shows how the computer's screen is organized.

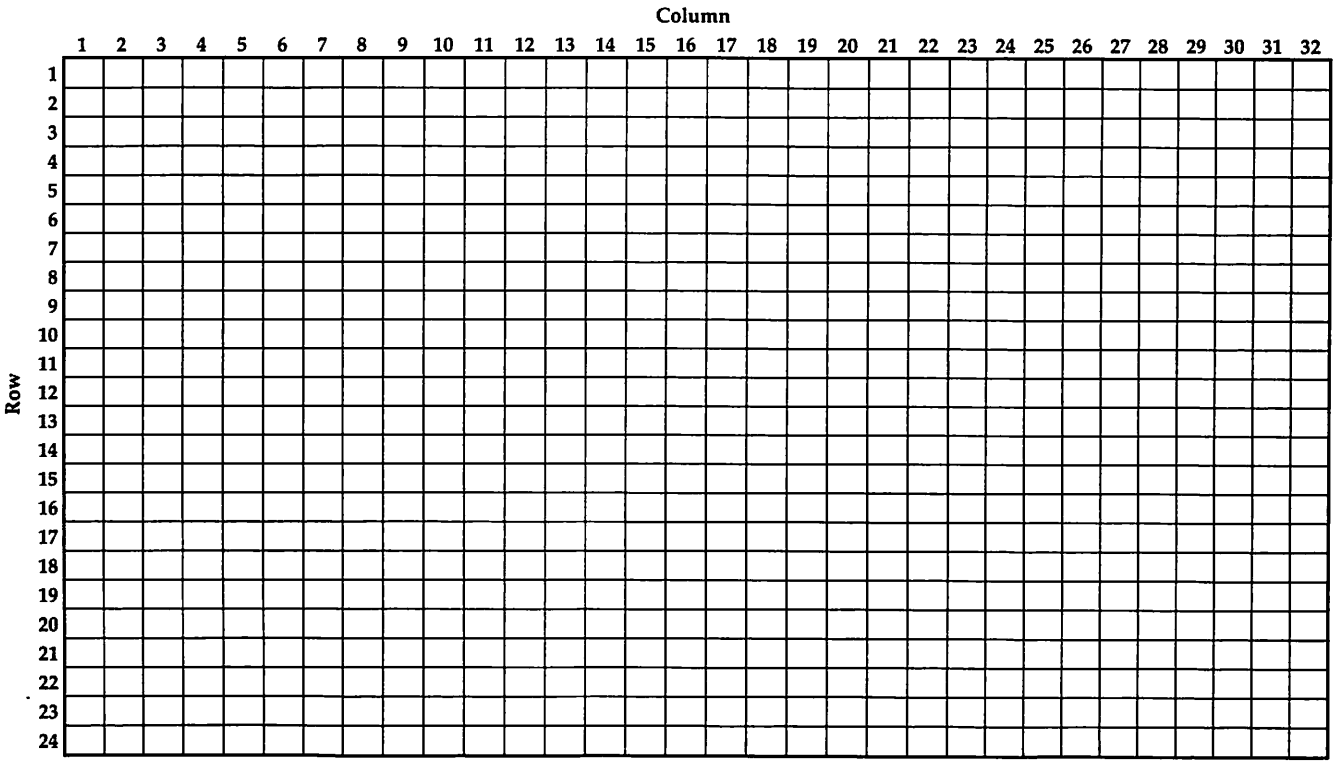
To use CALL HCHAR, you must give the computer three or four numbers to tell it where to put a particular character.

When using CALL HCHAR, the first number you type, after CALL HCHAR and a left parenthesis, is the row number. Follow that with a comma, and then the column number. Next you type the number which stands for the letter you want to appear on the screen. Finally, type a right parenthesis and press ENTER. Appendix A, "Characters," shows what number represents each letter, number, or symbol on the TI's keyboard. For example, the number 72 stands for the letter H.

ASCII Numbers. The computer uses these numbers to represent letters. These numbers are called ASCII codes; ASCII stands for the American Standard Code for Information Interchange. This is a code agreed upon by computer makers, so that an H is always represented by the letter 72, no matter what computer it is.

The TI has a command called ASC. If you want to see

Figure 2-1. TI Screen Map



what code value a letter, number, or symbol is represented by, just type ASC("*letter*") and press ENTER. Be sure to put the parentheses and quotes around the letter.

CALL HCHAR. Begin by typing:

```
CALL HCHAR (
```

but do not press ENTER yet. Be sure to type the left parenthesis (SHIFT 9).

Pick a row and column on the screen. Remember that the row must be between 1 and 32, and the column between 1 and 24. For this example, pick row 15 and column 25. Next, pick a letter, for instance, the letter A. If you look up A in Appendix A, you'll see that it's represented by the code value 65. Finally, type a right parenthesis (SHIFT 0).

Here's an example:

```
CALL HCHAR (15,25,65)
```

Now, press ENTER. The letter A will appear somewhere on the middle right of the screen.

More Than One Character. You can add one more number to the CALL HCHAR command to tell the computer to print more than one character at a time. If you want it to print 15 times, starting at the row and column you selected, all you have to do is add another comma after the ASCII code value and a number to represent the number of times you want it to print. Be sure to add the final parenthesis (SHIFT 0). For example, type:

```
CALL HCHAR (15,25,65,15)
```

When you press ENTER, a row of A's will PRINT, starting at row 15, column 25. Eight A's will print, and the other seven will spill over and show on the next line, starting at the far left. You could use the CALL HCHAR command to create almost any kind of horizontal display that you want. If you wanted a border at the top and bottom of the screen, for instance, all you'd have to do is enter these lines in a program:

Program 2-1. Border

```
5 CALL CLEAR
10 CALL HCHAR(1,1,95,32)
20 CALL HCHAR(24,1,95,32)
50 GOTO 50
```

This creates a border using the underline symbol, which is

ASCII code value 95. The top border is created by line 10, which starts at row 1, column 1, and puts 32 underline symbols across the screen. Line 20 does the same thing, but near the bottom of the screen. It begins at row 24, column 1, and places the symbols on the screen. Line 5 simply clears the screen so you can see the borders easier, while line 50 holds the characters on the screen. To break from this program, press the FCTN key and the 4 key at the same time.

Using VCHAR. You can also use a similar command, CALL VCHAR, which stands for *V*ertical *CHAR*acter. It works the same, except that the fourth number indicates how many times you want a character to be PRINTed down.

For example, type:

```
CALL VCHAR (15,25,65,15)
```

When you press ENTER, you'll see ten A's PRINT down, and another four starting at the next column and at the top row. Wait! Why do you have only 14 A's? Why not 15, the number specified in the command?

When the TI executes a line in the command mode, the screen moves up one row after the command is finished. You'll see how to avoid this later.

Using the CALL VCHAR command, you can complete the border by adding left- and right-hand lines. Add these lines to "Border."

```
3Ø CALL VCHAR(2,1,124,22)
4Ø CALL VCHAR(2,32,124,22)
```

You'll notice that the ASCII code value is different from the horizontal borders. ASCII code value 124 is used to create the vertical borders. Line 30 draws the left-hand border by starting at row 2, column 1 and PRINTing 22 symbols, just enough to fill in the gap between the top and bottom borders. The right-hand border is created by line 40, which PRINTs 22 symbols starting at row 2, column 32.

With only six lines, you've drawn a border. You can use this technique in any game that requires a playing field marked off from the rest of the screen.

You now have the most fundamental part of any arcade game. Every game you create on the TI computer will use CALL HCHAR or CALL VCHAR, because these commands are necessary to create the screen, the world where the game takes place. You'll use PRINT to put messages on the screen,

but your game won't *play* unless you can put your game characters exactly where you want them on the screen. That's what CALL HCHAR and CALL VCHAR do.

Of course, there are more things you'll learn before you create your own games. You'll learn how to move characters on the screen, both by computer and player control; how to tell if two characters collide; and how to make sound effects with your computer. This may seem like a lot, but you'll learn step by step, so that in the end, you'll know how to put it together to make a game of your own.

How a game looks is as important as how it plays. In the next chapter, you'll see how to transform single letters of the alphabet into graphic pictures that look like animals, people, spaceships, or almost anything you can draw. Not only that, but you'll be able to choose colors for your character pictures.

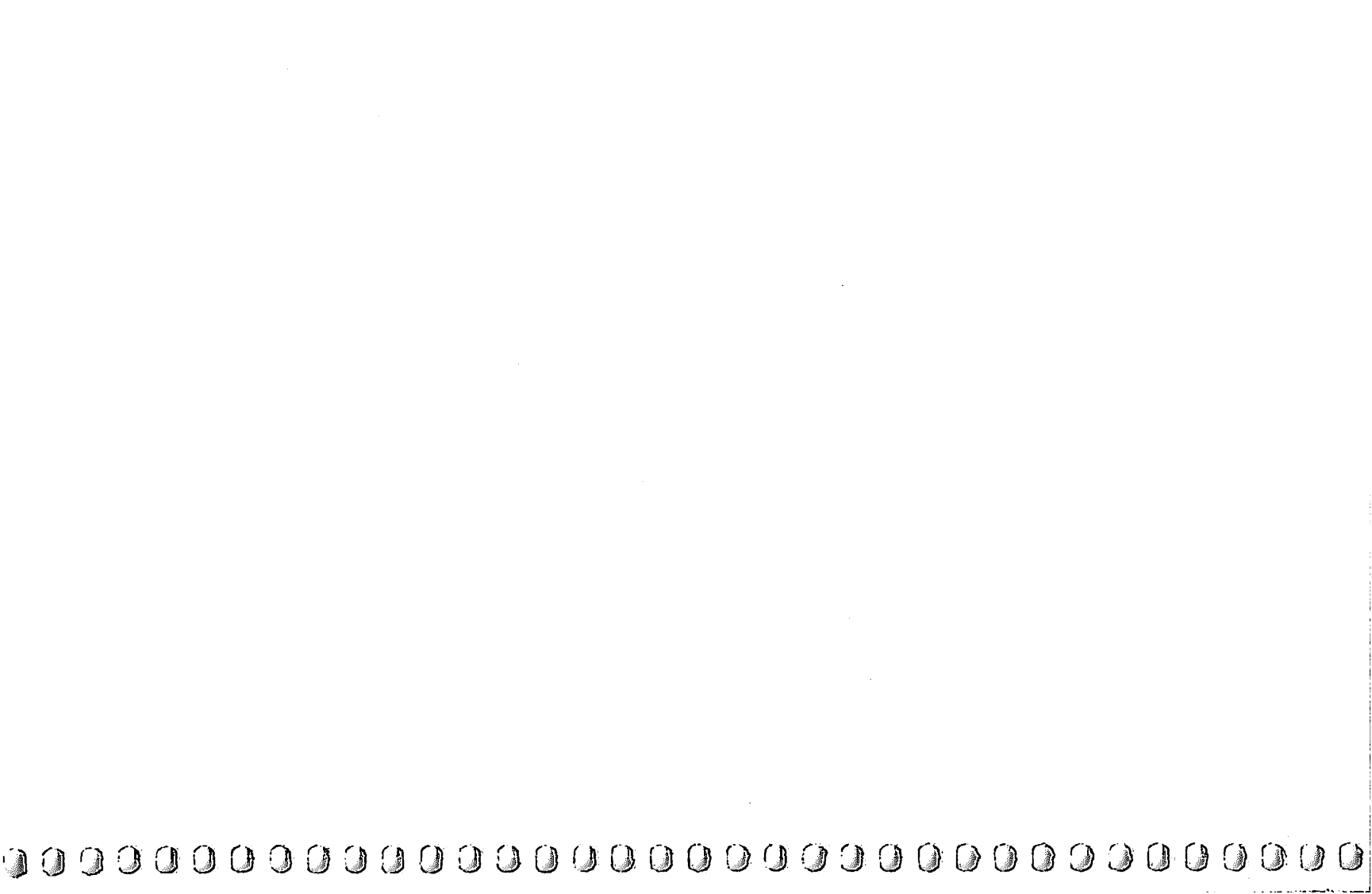
Review

In this chapter you've seen some of the building blocks that you'll use in programming arcade games on your TI-99/4A computer.

- By using PRINT, you can put letters on the screen.
- A program is a group of commands, each command having its own line number. RUN makes the computer execute the program. You can also use LIST and EDIT to make changes in your line-numbered commands.
- CALL HCHAR and CALL VCHAR place characters on the screen, exactly where you want them.

The form for HCHAR and VCHAR is:

CALL HCHAR or CALL VCHAR (*row, column, ASCII #, # of chars*)



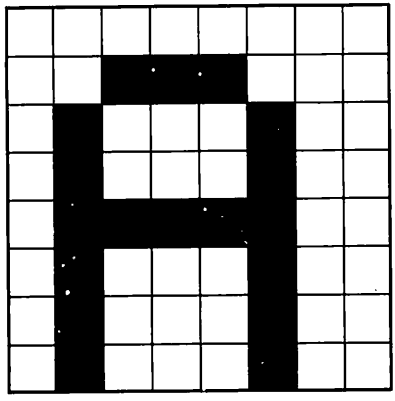
3 Defining Custom Characters

3 Defining Custom Characters

Your computer has the ability to change the shape and color of its characters. You're not restricted to just the letters, numbers, and symbols on its keyboard. You can use this to create creatures, objects, people, or backgrounds, or almost anything you want!

If you look closely at your TV screen, you'll see that each letter is made up of dots in a pattern eight dots wide by eight dots high. Each dot can be turned on or off—colored or not colored. Their arrangement creates the shape of the character. For example, the letter A would have a pattern like Figure 3-1.

Figure 3-1. Dot Pattern



To create your own custom game characters, you simply change the dot patterns. It's not even difficult.

CALL CHAR

The CALL CHAR command lets you create your own custom character by changing the dot pattern of any standard character. By using CALL CHAR, you can change the dot pattern of the letter A so that it resembles a spaceship. This is

called *redefining a character.* After you have redefined a character, its new dot pattern will be printed on the screen whenever you use it in a PRINT, CALL HCHAR, or CALL VCHAR statement in the program.

There are two codes that make CALL CHAR work. The first is simply the ASCII code value of the character, from 32 to 159, that you want to change. You can look at Appendix A to see which characters are represented by which ASCII values. For example, the space is ASCII 32, the ! (exclamation mark) is 33, the # (number sign) is 35, and so on.

The TI divides these numbers into two groups for CALL CHAR. When you use CALL CHAR to redefine the characters whose ASCII numbers are between 32 and 127, they will stay redefined *only* while your program is running. Then they will go back to their normal patterns. For example, if you change an A to a rocket ship, it will have that shape only while your program is running. After you stop the program, the letter A will come back. This is a handy feature because otherwise you might change a character that you need to enter. It can get confusing if you type A and a rocket ship appears on the screen.

If you prefer, you can use the characters that have the ASCII values 128-159, and they will stay changed until you change them back yourself or turn off the machine.

Either set of numbers is all right, but you might want to use only the values from 32-127 so that you won't accidentally use an earlier redefined character.

Select a Character Number. First of all, you must select a character you want to change. Usually you won't want to change the character whose ASCII number is 32, because that is the space key, what the TI uses for any part of the screen that doesn't have a character on it. You might also want to save the alphabet so you can print messages on the screen, like GAME OVER or NEXT ROUND.

Usually a good group of standard characters to change is the symbols which begin at ASCII 33 (!) and end at ASCII 47 (/). While you will use some of these in your program, they will not actually be redefined until the program runs, so that you can type and LIST your programs and still see, for instance, a +.

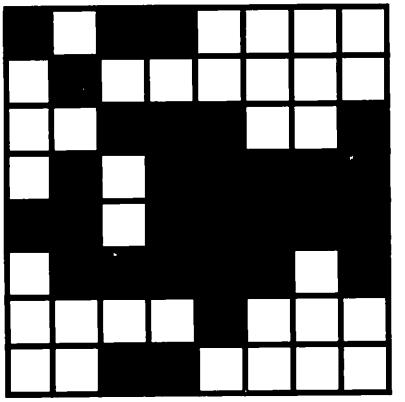
For an example of how to use CALL CHAR, we'll use the ! (exclamation point). Its ASCII number is 33.

Next, you have to decide what you want your new pattern to look like.

Making a Picture. Some graph paper will be handy for the next step. If you can find some that's marked off in blocks of eight by eight, that's even better. Otherwise, draw an eight-by-eight block on the graph paper, and create your figure in it, coloring in the squares that you want to appear *on* in the new character, and leaving blank the ones that you want *off*. When the custom character is placed on the screen, the *on* dots will show in one color, while the *off* dots will appear in the same color as the background.

For example, here's a picture of the head of a caterpillar:

Figure 3-2. Caterpillar Head



The caterpillar is facing left, and has a mouth, eye, nose, and feelers. You can change its shape it you want. The important thing is to draw a picture first, and put it into an eight-by-eight dot pattern.

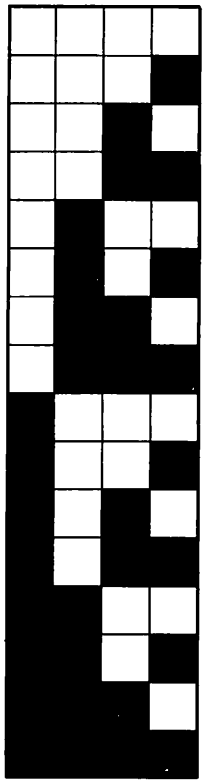
After you have created the dot pattern, you must put the pattern into a special code that the TI can read.

Think of each eight-by-eight block as if it were divided into two parts, each four dots wide. For each four-dot section, there are 16 possible combinations of *on* (colored) and *off* (not-colored) dots. In the TI, a binary code is used to specify which dots are on and which are off. In binary notation, a 0 stands for off and a 1 stands for on. However, you can use a type of shorthand, called hexadecimal, to indicate the combination of

colored dots in your own custom character.

Figure 3-3 shows each combination of on and off dots, as well as the corresponding hexadecimal codes.

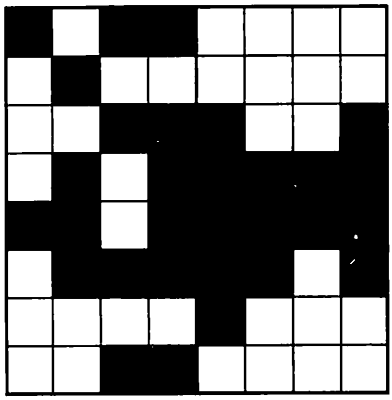
Figure 3-3. Character Combinations and Codes

Dot Combination	Hexadecimal Code
	0
	1
	2
	3
	4
	5
	6
	7
	8
	9
	A
	B
	C
	D
	E
	F

To create your own figure, you simply arrange the codes for the four-dot sections in order, beginning at the top-left corner and working to the right, then down to the next row, proceeding until you reach the bottom-right corner. If you remember that it's just like the way we read, starting at the top and moving to the right, you won't have any problem with it.

Make sure that the combinations match the code you've chosen. For instance, the caterpillar head would have these hexadecimal code values for its dot combinations:

Figure 3-4. Caterpillar Codes



The whole pattern for the caterpillar's head is:

B040395FDF7D0830

Once you have created the number and letter pattern that represents all the dots, you can put it into the CALL CHAR command.

Here's how it works:

- Type CALL CHAR.
- Enter a left parenthesis (SHIFT 9).
- Type the ASCII code value of the letter, number, or symbol you want changed.
- Enter a comma, followed by a quote mark (FCTN P).
- Type all 16 letters and numbers that make up your pattern, with no spaces in between. (If you type less than 16, the computer will fill the rest with blanks, and if you type more than 16, the computer will ignore the remainder.)
- Finally, type another quote mark, and a right parenthesis (SHIFT 0).

In the example of our caterpillar head, type:

```
CALL CHAR (96,"B040395FDF7D0830")
```

When typing this, be sure to put the parentheses, comma, and the quotes in the right places, or the TI won't know what you want it to do.

Putting It on the Screen. If you typed a CALL CHAR command by itself, nothing would happen. Actually, you

would see the screen blink, but the command's effect would be over before you were able to see it.

What you have to do is create a program to see it.

First of all, type NEW and press ENTER. This will clear your computer's memory.

Before you begin typing, make sure that your ALPHA LOCK key is pressed and locked down. Otherwise, you may have problems with some of the letters you type.

Then enter :

```
120 CALL CHAR(96,"B040395FDF7D0830")
```

However, if you run the program by typing RUN and ENTER, you still won't see anything. You must add a special line afterward to keep the picture you've created on the screen so that you see what has happened.

Type:

```
170 GOTO 170
```

GOTO is a command that tells the computer to go to a specific line and execute that line. If you are on line 170 and you tell the computer to GOTO line 170, the program will stay stuck on that line, always going to itself, until you stop it. If you don't have something like this, when the program ends and after it has processed the last command, the screen character shapes will go back to their original values. (See the section "Stopping the Program" in this chapter to see how to stop your program.)

You still can't see the new character's shape until you print it on the screen. To do this, use CALL HCHAR.

To put the caterpillar's head at row 5, column 10, for example, you would add:

```
150 CALL HCHAR(5,10,96)
```

This will put character 96, the character you just redefined to look like a caterpillar head, on the screen.

(Notice that you are able to type line 150 *after* you typed line 170. However, if you LIST the program, you'll see that line 150 has been inserted between line 120 and line 170.) It's a good idea to always use LIST to see if you've made any mistakes.

Before you type RUN, however, add one more command:

```
100 CALL CLEAR
```

This will clear the screen so that you will have an empty screen

when the program runs.

Now, when you type RUN and press ENTER, this short program will display a caterpillar's head on the screen.

Program 3-1. Caterpillar

```
100 CALL CLEAR
120 CALL CHAR(96,"B040395FDF7D0830")
150 CALL HCHAR(5,10,96)
170 GOTO 170
```

Stopping the Program. After you have typed RUN and seen your caterpillar, you will notice that pressing the keys on the keyboard has no effect. This is because line 170 has the computer stuck in what is called an *infinite loop*. Hundreds of times a second, the computer is executing line 170.

The only way to stop it, besides turning the computer off, is to press the FCTN key, hold it down, and press the 4 key. This should stop your program and you will see the message:

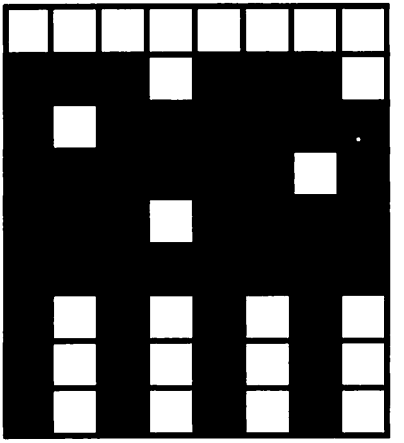
* BREAKPOINT AT 170

The computer will beep to tell you that it has stopped.

Also, you'll notice that your caterpillar's head has changed back to a single quotation mark, because it kept its redefined shape only while the program was running.

Adding Legs and a Body. Your caterpillar might like a body and legs. All you have to do is design a character to look like a body segment. Here's how it might look, complete with dot combination codes:

Figure 3-5. Body and Legs



3 Defining Custom Characters

Add this line to the program we've been creating, Program 3-1:

```
130 CALL CHAR(97,"ØØEEBFFDEFFFAAAA")
```

To put the character on the screen, add this line:

```
160 CALL HCHAR(5,11,97,5)
```

This will put character 97, which you redefined in line 130, into row 5, column 11. Column 11 was picked because it is one column to the right of the caterpillar's head. The fourth number, 5, is the number of times that the character will be repeated to make a long body.

When you type RUN and press ENTER, you'll see a caterpillar, complete with head and body. You can make the body longer or shorter by selecting a different value for the fourth number in the CALL HCHAR command.

Adding Color

So far in this chapter, you've seen how to create new shapes. But there's another way you can make your characters unique, and that's to add color. There are two ways to add color to the screen. The first is by changing the whole color of the screen, and the second is to give each character, or group of characters, a different color.

Screen Color. Depending on how your TV screen is adjusted, you should see a green background when your program runs, and a blue background when you are typing.

If you would like to change the color of the background screen, you use the CALL SCREEN command:

CALL SCREEN (*n*)

where *n* is a number from 1 to 16. Each number represents a different color. For a list of the colors and their number values, refer to Appendix B, "Color Values." You could retype the CALL SCREEN command 16 times to see each color, but an easier way is to enter and RUN this short program. You'll see the color and its value for a moment on the screen.

Program 3-2. Screen Color

```
10 FOR COLOR=1 TO 16
20 CALL CLEAR
30 CALL SCREEN(COLOR)
40 PRINT "COLOR VALUE";COLOR
50 FOR DELAY=1 TO 1000
```

```
60 NEXT DELAY
70 NEXT COLOR
80 CALL CLEAR
90 GOTO 10
```

As you can see from looking at the color value table in Appendix B, color 1 is transparent. If you had a transparent graphics character, it would be the same color as the screen. In other words, it would be invisible. Specifying CALL SCREEN (1), however, will make the screen black. Since the print on the TI is displayed in black too, you'll not see a message for the black color in this program. In fact, because the transparent screen color prints in black, the first two seconds of this program will create a black screen.

Changing the screen color of our caterpillar program could be done by simply adding this line:

```
110 CALL SCREEN(16)
```

which will turn the screen color to white. Of course, you could select a different color by entering another color value inside the parentheses.

Character Color. You can also change the color of any standard or custom character. All you have to do is use the command CALL COLOR.

The TI divides all the characters that it uses into groups of eight characters, called *character sets*. For example, the characters that have ASCII numbers from 32 to 39 are all known as character set 1, the characters that have ASCII numbers 40-47 are set 2, and so on. Appendix A "Characters," also lists the characters by set.

When you change a color, the entire set of eight characters changes. For example, if you change character set 1, you change the colors of the space, as well as those of the !"#$%&' symbols.

To use CALL COLOR, you must specify the character set number, the foreground color, and the background color. The foreground color sets the *on* dots in the character pattern, and the background color sets the *off* dots. If you are using a character set that does not include the space, character set 2, for instance, the background might be different from the space color, so that you can have multicolored characters.

The form for CALL COLOR is:

CALL COLOR (character set number, foreground color value, background color value)

For example, adding this line to Program 3-1, "Caterpillar," changes the character colors:

```
140 CALL COLOR(9,9,16)
```

This colors all of character set 9, so that the foreground color is 9, or medium red, and the background color is 16, which is white.

If you use color 1, which is called transparent, then whatever the screen color is becomes the color for background or foreground color.

In this example, you would get the same colors if you typed:

```
140 CALL COLOR(9,9,1)
```

because the screen color was previously set to 16.

You can change the character colors, just as you changed the screen color, by selecting another color value.

Here's the caterpillar program so far:

Program 3-3. Caterpillar Complete

```
100 CALL CLEAR
110 CALL SCREEN(16)
120 CALL CHAR(96,"B040395FDF7D0830")
130 CALL CHAR(97,"00EEBFFDEFFFAAAA")
140 CALL COLOR(9,9,16)
150 CALL HCHAR(5,10,96)
160 CALL HCHAR(5,11,97,5)
170 GOTO 170
```

Flickers. You can even use the CALL COLOR command to create flickering or blinking custom characters. To make the caterpillar figure flicker rapidly, for example, you could use the transparent color 1, alternating it with another character color. Replacing line 170 in Program 3-3 and adding the new lines 180 and 190 would do this. The changes you would enter are:

```
170 CALL COLOR(9,1,1)
180 CALL COLOR(9,9,16)
190 GOTO 170
```

RUN the program to see the difference this feature can make in your character's appearance. The character is made invisible by line 170, and then visible in line 180. It all happens so quickly, however, that it seems to flicker.

Disappearing Characters. You can make the character blink off and on at a slower pace by making a few more

changes. The character is again made invisible and then visible, only this time the delay loop in line 180 slows the process down. Here are the lines you need to change and add to the original version of Program 3-3 to see this happen:

```
170 CALL COLOR(9,16,16)
180 FOR DELAY=1 TO 100
190 NEXT DELAY
200 GOTO 140
```

Solid Characters. There are two ways you can create solid squares on the TI screen. These kinds of figures can be used in games to draw borders, walls of mazes, or simple playing fields.

One way to draw solid figures in color is to assign the same color to both the foreground and the background in the CALL COLOR command. This will make the eight-by-eight dot pattern one color, no matter what the on-off combination of dots. The only thing you should remember is that if you're using character set 1, which includes the space (ASCII code value 32), most of the screen will change to the new color. If you want to create solid squares of color, you should not use set 1.

As an example, let's change the caterpillar figure into a six-block bar. It's easy. All you have to do is change line 140 in Program 3-3:

```
140 CALL COLOR(9,9,9)
```

All the characters in set 9 now display in red, and the caterpillar becomes a solid red bar. The rest of the screen remains white. If this had been set 1, almost all of the screen would have changed to red, since much of the screen is filled with spaces.

The other way to create solid blocks, of course, is to make custom characters defined as 0. Although the character may seem empty, all you have to do is set the figure's foreground and background colors to the same color value, as you did with the caterpillar. If you want to type more on the keyboard, you can draw a character as completely filled by entering FFFFFFFFFFFFFFFF (16 F's). You would only have to specify the foreground color, then; the background could be set to anything and the square would still show on the screen.

But drawing characters is only part of the technique needed to create your own arcade games. Another is motion.

3 Defining Custom Characters

Even if you have a dazzling screen and unique characters, it still isn't a game unless the characters on that screen move and interact. The next chapter will show you how to do that.

Review

In this chapter you've learned how to make custom characters by using CALL CHAR, which defines a dot pattern eight-by-eight.

- CALL CHAR uses the ASCII character number and a set of numbers and letters that represents the dot picture. The form for the command is:

  CALL CHAR (*character number, "dot pattern code"*)

- CALL HCHAR PRINTs the custom characters on the screen.
- CALL COLOR can color your characters in the following manner:

  CALL COLOR (*character set number, foreground color number, background color number*)

Each set of eight characters has a character set number, from 1 to 16, and each color has a number, from 1 to 16.

4 Beginning to Move

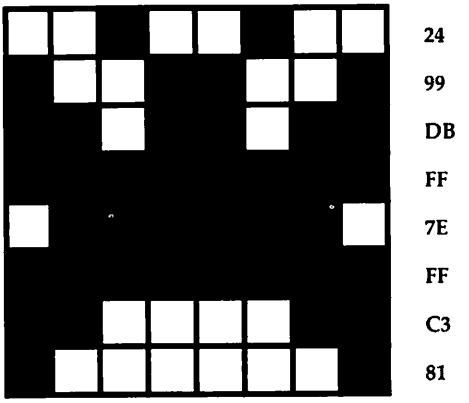
4 Beginning to Move

Now that you've seen how to redefine characters, it's time to see how to make them move. Making figures move is what arcade games are all about—if it doesn't move, you don't really have a game. This chapter will cover computer-controlled movement—making the TI move your standard or custom characters. The next chapter will explain how you can use the keyboard or joystick to move figures under player control.

First of all, you'll need a figure for the computer to move on the screen. Instead of a caterpillar, let's change it into a butterfly.

Here's a dot pattern for a butterfly.

Figure 4-1. Butterfly



To create the butterfly in Figure 4-1, type:

```
10 CALL CHAR(33,"2499DBFF7EFFC381")
```

This creates a figure as character 33, which is the exclamation mark.

More about Color. You'll notice that some color combinations work better on your TV set than others. A combination that will work well for your butterfly figure is an orange character on a black background. To see this, you'd enter:

```
20 CALL SCREEN(2)
```

to make a black screen, then

```
30 CALL COLOR(1,7,1)
```

to make the orange character.

The first 1 refers to character set 1, which includes the exclamation mark (ASCII 33). The 7 is actually dark red, but it looks orange on most TV sets when they are properly adjusted.

(When you are setting up your color TV and your TI computer, adjust the TV set so that the flesh tones look right, then switch back to the channel that your computer will be on. This way, you can use colors in a way that will match what other people will use.)

The third number in the CALL COLOR command is a 1, which means that the background of the butterfly's eight-by-eight grid will be transparent. As mentioned in Chapter 3, this means that it will match whatever the screen's color is.

To make sure you have an empty screen, it's a good idea to always type:

```
5 CALL CLEAR
```

Now, if you would like to see your butterfly on the screen, enter

```
40 CALL VCHAR(5,10,33)
900 GOTO 900
```

Line 40 puts character 33 in row 5, column 10. Line 900 is there so that your program will keep running.

The program looks like this so far:

Program 4-1. Butterfly

```
5 CALL CLEAR
10 CALL CHAR(33,"2499DBFF7EFFC381")
20 CALL SCREEN(2)
30 CALL COLOR(1,7,1)
40 CALL VCHAR(5,10,33)
900 GOTO 900
```

Making It Move

To make something *appear* to move on the screen, you'll go through a four-step process:

- You draw it on the screen, by using CALL CHAR to create your character in memory, and CALL VCHAR or CALL HCHAR to put it on the screen.

- Then you create a short delay. You need a delay because the image must appear on the screen long enough for you to see it. You create a delay simply by using a FOR/NEXT loop.

A FOR/NEXT loop is made up of two parts. The FOR portion of the loop would be written something like this:

```
FOR X = 1 TO 500
```

This sets up a counting system for keeping track of how many times X (or any variable) executes.

The NEXT part of the loop would be written as:

```
NEXT X
```

This sends the computer back up to the line that has the FOR portion of the FOR/NEXT loop. The FOR/NEXT cycle will continue as many times as specified in the statement. This type of loop slows the computer down.

Another use of FOR/NEXT loops is to cause a certain kind of action to happen a certain number of times. The first number after the FOR is the starting number of the variable (X in this case), and the second number is the final number. If you want it to count in increments larger or smaller than one, you can add the word STEP and the number you want to have it increment by. For example, you could enter something like this:

```
FOR X = 500 TO 1 STEP -1
```

and the computer would count backwards by ones.

Often you won't need to use a FOR/NEXT loop when simulating motion, because you'll put necessary calculations in the part of the motion cycle that comes between when a character is put on the screen and when it is erased. These calculations will slow the computer down.

- Next you need to erase your character at its present location before you move it to a new one. If you don't, it will look like it's leaving a trail behind. You can erase a character by simply using CALL VCHAR or CALL HCHAR, and using character 32, which is the space. Printing a space on top of anything erases it.
- Finally, you put your character at its new location with CALL VCHAR or CALL HCHAR, using a new row and column value for its new location.

These four steps will create motion on your TI computer.

4 Beginning to Move

Here's a sample program which will show you how to make your butterfly character move. To Program 4-1, add the following lines.

Program 4-2. Butterfly Motion

```
100 FOR X=1 TO 10
110 CALL VCHAR(5,10+X-1,32)
120 CALL VCHAR(5,10+X,33)
130 FOR Y=1 TO 100
140 NEXT Y
150 NEXT X
```

These six lines are very important for understanding how motion works. The following explanation will help to clarify these six lines, as well as the lines you typed in earlier.

Here is how the four steps of motion, mentioned earlier, work.

Step 1. The character was first PRINTed in Program 4-1, line 40. But each time it's printed again, line 120 will place it at its new row and column number. Line 120 uses the variable X, which was set up in the FOR/NEXT loop in line 100. X will increase by 1 each time the program goes through the loop, starting at 1 and going to 10. Because of this, the computer can use X and add it to the original column, 10. The first time through the loop, line 120 will print the character at column 11 (10+1=11), the next time around the loop it will print at column 12, and so on.

Step 2. The delay must come after the character is printed, but *not* before the character is erased and a new one is printed. Lines 130 and 140 use another FOR/NEXT loop, which creates a delay as the computer goes through the loop 100 times. You can, of course, make the Y loop longer or shorter by increasing or decreasing the range of values. This will lengthen or shorten the delay time.

Step 3. Next you erase the old character in line 110 by PRINTing the space character (32) at the previous location. This also uses the X in the loop, but subtracts 1 from it, since you want to erase the *old* character. Since X starts out at 1, the first time through the loop, the erasing will be at 10+X−1 (this could also be stated as 10+1−1). Since 1−1 is 0, it will erase at column 10. The next time through the X loop, X will be 2, so 10+X−1 will be 10+2−1, or 11. The erasing is always one column (or row, if it is moving up or down) behind where it PRINTs.

Step 4. Finally, you PRINT the new character in line 120. This must happen immediately after the erasing, with no delays of any kind in between. Line 120 uses X to put character 33 in a new column. Moving usually involves a cycle of putting characters on the screen, delaying, erasing, and putting them on the screen at a new position.

You can have the character move up or down by adding numbers to the row number instead of the column. For example, change lines 110 and 120 to read:

```
110 CALL VCHAR(5+X-1,10,32)
120 CALL VCHAR(5+X,10,33)
```

When you type RUN and press ENTER, your butterfly will move downward instead of to the right.

By changing a few lines, you can make the butterfly move back and forth across the screen in a continuous loop. What you end up doing is creating another set of instructions to move the character from right to left, erasing the previously printed figure and reprinting it in the column to the left. Most of these lines are duplications of ones you've already entered.

To move the butterfly back and forth, simply add these lines to Program 4-2:

```
160 FOR X=10 TO 1 STEP -1
170 CALL VCHAR(5,10+X+1,32)
180 CALL VCHAR(5,10+X,33)
190 FOR Y=1 TO 50
200 NEXT Y
210 NEXT X
220 GOTO 100
900 GOTO 900
```

Line 160 moves the character from right to left by assigning X a value starting with 10, so that the butterfly's position is at column 20 (10+10), its last location as it moved left to right. Each time through the main movement loop of the program, X decreases by 1, due to the STEP −1 statement. Lines 170 and 180 first erase the previous character and then print a new figure to the left. (The second time through the loop, the butterfly shows up in column 19 [10+9]; the third time through the loop, in column 18; and so on.)

Line 190 creates a delay loop, this time half as long as when the character moved from left to right. Notice how much faster the butterfly moves from right to left, compared to left to right. Lines 200 and 210 are the NEXT part of the FOR/NEXT

loop begun in line 160, while line 220 simply sends the program back to line 100, where the movement from left to right executes again.

This movement will continue until you press the FCTN and the 4 keys.

You could use something like this in a game of your own if you wanted a target that constantly moved from side to side, like the ducks in a carnival gallery. To move the butterfly up and down, over and over, all you'd have to do is change lines 110 and 120 as you saw earlier, and then alter lines 170 and 180 to:

```
170 CALL VCHAR(5+X+1,10,32)
180 CALL VCHAR(5+X,10,33)
```

If you put a number of characters on the screen, each moving in its own particular pattern, you could have the basis for a simple arcade-style target-and-shoot game.

Random Motion

Often it's useful to have a character move in ways that the player cannot predict. If you know that a character will always move from left to right, for instance, the game may be too easy.

Your TI has a function called RND, which picks a number that will be different each time. The number selected is always a decimal function between 0 and 1. For example, it could be .5, .33, .76934, or .102113.

This number by itself might not seem too useful, but you can multiply it by any number to get a larger one. For example, if you multiply it by 10, you'll get random numbers between 0 and 9. They'll still have several numerals after the decimal point, however.

To use RND effectively, you must use another function, called INT, which stands for INTEGER. This function knocks off any fractional part of a number. For example, if you have a number like 1.33, it will drop the .33, leaving you with the integer 1.

Because RND picks numbers between 0 and 1, if you use the INT command which rounds off fractions, you would always end up with 0. You must be sure to multiply the RND function inside the parentheses after the INT command. To move up the range of possible numbers, you need to *add* a number to the end of the statement.

Here's a formula you can use.

To get a random number between 1 and X, use:

INT (RND * X) +1

For example, to get a random number between 1 and 32, which would be helpful in selecting a column on the screen at random, the formula would look like this:

INT (RND * 32) + 1

If the *+1* were omitted, the range would be between 0 and 31.

Whenever you use this formula in your program, you'll get a different number between 1 and 32. (The * character is the multiplication sign.)

A Fluttering Butterfly. To set up a program to move the butterfly in random motion, in any one of eight directions (up, down, left, right, or up-right, up-left, down-right, down-left), enter the following new program:

Program 4-3. Random Butterfly

```
10 REM  BUTTERFLY
20 CALL CLEAR
30 CALL SCREEN(2)
40 CALL COLOR(1,7,1)
50 OR=12
60 OC=16
70 CALL CHAR(33,"2499DBFF7EFFC381")
100 REM  LOOP
110 X=INT(RND*3)-1
120 Y=INT(RND*3)-1
130 NR=OR+X
140 NC=OC+Y
150 CALL VCHAR(OR,OC,32)
160 CALL VCHAR(NR,NC,33)
170 OR=NR
180 OC=NC
200 GOTO 100
```

This program can be divided into two parts:

- The initial part, which sets up the beginning values of the program. This includes lines 10-60, which create the shape, colors, background, and initial position of the butterfly.
- The part of the program that moves the butterfly. In this part, the numbers used to calculate the new position of the butterfly are created, the rows and columns of both the new and old position of the butterfly are calculated, the old butterfly is erased, and the new butterfly is created. Finally, the old row and column become the new row and column,

so that the process can start over again. Notice that there is no FOR/NEXT loop to create a delay, as there was in the last program. This is because there will be enough delay in the calculation of the new and old row values. You could still add a delay if you want, because one of the factors of game design has to do with how long an image appears on the screen. If it is too quick, the eye doesn't pick up all the details, and if it is too slow, the player gets bored.

Here is a more specific line-by-line explanation.

Line 10 is a REMark; anything following REM will be ignored by the computer, but it's often important for you to know what a program does. Adding remark labels will make your programs easier to follow and is a good habit to get into, especially in more complicated programs where you may want dozens of REM statements to label all the different parts.

Line 20 clears the screen, line 30 sets the screen color to black (2), and line 40 sets the color of the character to orange, with a transparent background.

Lines 50 and 60 establish the variables OR and OC (*O*ld *R*ow and *O*ld Column). The starting position is row 12 and column 16, which is roughly at the center of the screen.

Line 70 creates the butterfly using CALL CHAR.

Line 100 is another remark to show that you have started the main part of the program. Line 100 is the start of the main movement loop, with line 200 as its other end. The program will go back and forth between line 100 and line 200.

Lines 110 and 120 set up random numbers. What you want to do is get numbers that are either 1, 0, or −1, for both the row and column *change*. INT (RND*3) will produce numbers between 0 and 2, and subtracting 1 will make them between −1 and 1. X will be used for the row change, and Y will be used for the column change.

Lines 130 and 140 calculate the new row and columns that you want to print the butterfly at. NR and NC are the *N*ew Column and *N*ew *R*ows. New and Old are used to show the positions where the butterfly will be printed (New) and where it will be erased (Old).

Lines 150 and 160 use CALL VCHAR to first erase the old butterfly by putting character 32 (space) at row OR and column OC. Then, character 33 (butterfly) is put on the screen at row NR and column NC.

Lines 170 and 180 change the *values* of the old row and

column to the new row and column. Only by using this will the character erase when line 150 is executed the next time through the main loop.

Line 200 simply sends the program back to line 100, so that the loop repeats.

Checking for the Edge of the Screen. When you run the program, you can watch the butterfly fly around the screen, but after a while, the program will stop and you'll see an error message, probably one that reads BAD VALUE IN LINE 160.

This is because the butterfly fluttered too close to an edge and the NR and NC values were higher or lower than the TI allows. Since the row numbers can be only from 1 to 24 and the column numbers from 1 to 32, if your program has a row number greater than 24 or less than 1, or a column number greater than 32 or less than 1, you will get the BAD VALUE error statement from the TI.

To avoid this, you must add a way to check for row and column values. The easiest way to do this is to add a GOSUB command, which will make the program go to a subroutine, do what is there, and then RETURN to the main program. You could add the subroutine to the main program, but it's often easier to understand if you have a simple main loop, and use GOSUB commands to process more complicated kinds of information by sending the computer out of the main loop to a subroutine and then having it return. This way, you can use the same subroutine over and over again without cluttering up your main loop or having to rewrite the lines. For our example, simply add the line:

```
145 GOSUB 300
```

This will check the NR and NC values that were just created in lines 130 and 140.

Now add:

```
300 REM  CHECK FOR SIDES
310 IF NR<1 THEN 400
320 IF NR>24 THEN 400
330 IF NC<1 THEN 400
340 IF NC>32 THEN 400
350 RETURN
```

This subroutine has the REM statement CHECK FOR SIDES in line 300. Lines 310 to 340 test NR and NC to see if they are numbers that can be used by the TI for the CALL VCHAR routine. If they are smaller or greater than the

numbers that are legal, the program will jump to line 400. Otherwise, the program will continue.

Line 350 is a RETURN command. RETURN must always be the last line in a GOSUB subroutine, and it will make the program go back to the last GOSUB command.

The subroutine to reset the values for NR and NC look like this:

```
400 REM  ERROR
410 NR=12
420 NC=16
430 RETURN
```

This is a separate part of the same subroutine that the program will shift to if any boundary errors are found in lines 310-340.

Lines 410 and 420 change the new row and column numbers from the edge of the screen to numbers that will be at the center of the screen. Line 430 RETURNs the program to line 145, and the butterfly continues to move.

The complete program to move your butterfly character randomly around the screen looks like this:

Program 4-4. Random with Edge Checking

```
10 REM  BUTTERFLY
20 CALL CLEAR
30 CALL SCREEN(2)
40 CALL COLOR(1,7,1)
50 OR=12
60 OC=16
70 CALL CHAR(33,"2499DBFF7EFFC381")
100 REM  LOOP
105 RANDOMIZE
110 X=INT(RND*3)-1
115 RANDOMIZE
120 Y=INT(RND*3)-1
130 NR=OR+X
140 NC=OC+Y
145 GOSUB 300
150 CALL VCHAR(OR,OC,32)
160 CALL VCHAR(NR,NC,33)
170 OR=NR
180 OC=NC
200 GOTO 100
300 REM  CHECK FOR SIDES
310 IF NR<1 THEN 400
320 IF NR>24 THEN 400
```

```
330 IF NC<1 THEN 400
340 IF NC>32 THEN 400
350 RETURN
400 REM  ERROR
410 NR=12
420 NC=16
430 RETURN
```

Another way to program this kind of movement is to use the conditional OR to check if the character is close to a screen edge. This method will shorten the program somewhat by checking for two different things in one line. It would look like this:

Program 4-5. Side Checker with OR

```
10 REM  BUTTERFLY
20 CALL CLEAR
30 CALL SCREEN(2)
40 CALL COLOR(1,7,1)
50 OR=12
60 OC=16
70 CALL CHAR(33,"2499DBFF7EFFC381")
100 REM  LOOP
105 RANDOMIZE
110 X=INT(RND*3)-1
115 RANDOMIZE
120 Y=INT(RND*3)-1
130 NR=OR+X
140 NC=OC+Y
145 GOSUB 300
150 CALL VCHAR(OR,OC,32)
160 CALL VCHAR(NR,NC,33)
170 OR=NR
180 OC=NC
200 GOTO 100
300 REM  CHECK FOR SIDES
310 IF (NR<1)+(NR>24)THEN 400
320 IF (NC<1)+(NC>32)THEN 400
330 RETURN
400 REM  ERROR
410 NR=12
420 NC=16
430 RETURN
```

Lines 105 and 115 were added, lines 310-330 were changed, and lines 340 and 350 were eliminated. Those are the only differences between this and Program 4-4.

Lines 105 and 115 contain a new command, RANDOMIZE.

This will create truly random numbers, which the RND function does not actually do. If you RUN a program which has the RND function over and over, you would see the same sequence of numbers again and again. This makes a game using the RND function seem to follow a certain pattern, something you probably don't want. You want each game to be a little different from the last.

To generate true random numbers, it's a good idea to place the RANDOMIZE command somewhere in your program. Some programmers put it only near the beginning of the program, but it's probably best to use RANDOMIZE just before you use a statement with a RND function. This is what lines 105 and 115 do.

Lines 310 and 320 may look confusing, but they're really quite simple to understand. You can use the + sign in an IF/THEN statement to simulate the logical OR other computers allow. The logical OR means that at least *one* of the parts of the line (before and after the + sign) must be true to shift the program. For instance, in line 310, if *either* NR<1 or NR>24 is true, the program shifts to the subroutine at line 400. Only *one* of those conditions has to be met for the program to move to the subroutine. Notice that the parentheses are used to separate the different conditions from the + sign. This eliminates any possibility of the computer getting confused and thinking that the value 1 is to be added to something.

In other situations, you may want to have *both* sections of the line be true before a shift to another line takes place. You can do this by using the logical AND, which is represented on the TI by the * sign in an IF/THEN statement. For example, if you wanted the subroutine at line 400 called only when *both* NR<1 and NC>32 are true (in other words, only when the butterfly is in the upper-right corner of the screen), you could use a line such as:

```
IF (NR<1) * (NC>32) THEN 400
```

Both parts of the line must be true for the program to shift to the subroutine. You could use this to call a subroutine congratulating you on winning the game, for example, if the object was to get the butterfly from one corner of the screen to another, perhaps avoiding flower obstacles on the way.

Character motion such as patterned and random movement will be necessary for your arcade games because usually

you'll want to create some moving character for your game player to react against.

However, there is one more kind of motion which is absolutedly essential to arcade game play, and that is the motion that happens when the player provides input and tells the computer what to do by pressing a key or pulling a joystick.

In the next chapter you'll see how to have your player's wishes transferred to the computer program.

Review

In this chapter you saw how to create motion on the screen by moving a butterfly character around in both patterned and random ways.

The four steps to create movement are:

- Using CALL VCHAR or CALL HCHAR, you place a character on the screen. You use CALL CHAR to create you own custom characters.
- A FOR/NEXT loop can be used to create a short delay so that the characters can be more easily seen. Sometimes the main loop's calculations do this for you.
- The character has to be erased at its old row and column location. Printing a space character (ASCII 32) does this.
- Finally, the character is displayed on the screen at a new row and column position. It's important that there be no delay between the erasing and putting a new character on the screen, or you will have the appearance of two characters on the screen.
- It is also important to check that the row and column numbers are within the limits of what the CALL VCHAR command allows. Row numbers must be between 1 and 24, and column numbers between 1 and 32. Subroutines can check for errors.

5 Keyboard and Joystick Control

5 Keyboard and Joystick Control

In the last chapter you saw how to create motion by moving a butterfly character. But simply moving things on the screen is usually *not* a game: To have an arcade game, the player should be able to interact with the game's imaginary world.

The player should have some way to affect the computer program. You can do this in two ways: the TI's keyboard, or joysticks.

Using the Keyboard

The TI computer accepts commands from the keyboard when you use one of two different BASIC keywords.

INPUT. The first way to use the keyboard is with the INPUT command, but it really works too slow for player-controlled movement. INPUT accepts as many keystrokes as the player types in until the ENTER key is pressed. A use for this might be to have the player type in his or her name. For example:

```
10 INPUT "WHAT IS YOUR NAME?":A$
```

would have the computer first display the phrase WHAT IS YOUR NAME?, and then it would store as A$ whatever was entered. A$ could be printed later, perhaps at the end. For example:

```
PRINT "CONGRATULATIONS, "; A$ ;"! YOUR SCORE WAS
"; SCORE ; "POINTS."
```

The player's name would appear in the message.

This is a nice way to embellish your game, but it's an extra, not a necessary ingredient for a fast-action game.

Because the program stops to receive information when you use INPUT, it's not as useful in a game situation as the other keyboard command, CALL KEY.

CALL KEY. You can use the command CALL KEY, which will look to see if a key has been pressed, to receive player input.

CALL KEY has three parts to it.

Key-unit. The first part of the CALL KEY command is called the *key-unit* and tells the TI how to refer to its keyboard.

You can use this to split the keyboard in two, look for upper- and lowercase, or use a forced uppercase mode.

In this book, the programs will usually use a key-unit of 3, which does three things:

- All input, whether it is lowercase or uppercase letters, will be interpreted as uppercase letters. This is a safety factor, because then the player doesn't have to worry about whether the SHIFT key or ALPHA LOCK key is pressed.
- The key-unit of 3 also changes the function keys (which are produced by simultaneously pressing the FCTN key and another key) to specific numbers. This will prevent a program being erased if the FCTN and the = keys are accidentally pressed at the same time.
- The key-unit of 3 also ignores any control characters (which are produced by simultaneously pressing CTRL and another key). This insures that the player won't accidentally create a code that your game won't understand.

Return-variable. The second part of the CALL KEY command is called the *return-variable*. This can be any variable you choose. When CALL KEY is used, the computer will put the number of any pressed key into this variable. For example, if you specify that the return-variable of the CALL KEY is *K*, the ASCII number of a pressed key will be stored in the variable *K*. You can use this variable to make later game decisions.

For example, if you use *K* as the return-variable, and the A key was pressed, you'll find that K has the number 65 stored in it. If you use the key-unit of 3, it won't matter whether you press A or SHIFT A; you'll still get the number 65. Refer to Appendix A, "Characters," for the values for each key.

If no key is pressed, the number −1 is put into the return-variable.

Status-variable. The third part of the CALL KEY command is the *status-variable*. This can be any variable you choose. For example, you could call it *S*. When CALL KEY is used, a number will be put into the status-variable that will tell you one of three things:

- If the number is 1, you'll know that a new key was pressed since the last time CALL KEY was used. This can be helpful if, for example, you want to keep the player from constantly pressing the same key.
- If the number is 0, you'll know that the same key was

pressed as the last time CALL KEY was used. This could be used to wait until a new key is pressed.

- Finally, if the number is −1, you'll know that *no* key was pressed. You could use this to wait for any key to be pressed.

Motion

While any keys could be used, you'll probably find it easiest for the player to use the E, X, S, and D keys to make something move. This is because arrows are printed on the *sides* of these keys. The *E* key is up, the *X* key is down, the *S* key is left, and the *D* key is right.

By using CALL KEY and testing the return-variable, you can use the ASCII values that the return-variable contains to move your character around. Here are the ASCII values for the arrow keys:

Letter	ASCII Value	Arrow
E	69	up
X	88	down
S	83	left
D	68	right

Putting CALL KEY Together. To use CALL KEY, you need the key-unit, the return-variable, and the status-variable in the command. Here's the form of the CALL KEY command:

CALL KEY (*key-unit, return-variable, status-variable*)

The key-unit is a number from 0 to 5; we'll usually use 3. The return-variable and status-variable can be any variable.

Here is a program that demonstrates how to move a character with CALL KEY. It's divided into three parts:

- The beginning, which sets up the variables and gives everything a starting value.
- The main loop, which checks to see if a key has been pressed.
- A smaller section which the computer goes to if something happens. This is often a subroutine.

Often you will find that arcade game programs are set up in this way. You must set things up, wait for something to happen, and then make it happen.

The Setup. To set up the variables and character, you could enter:

5 Keyboard and Joystick Control

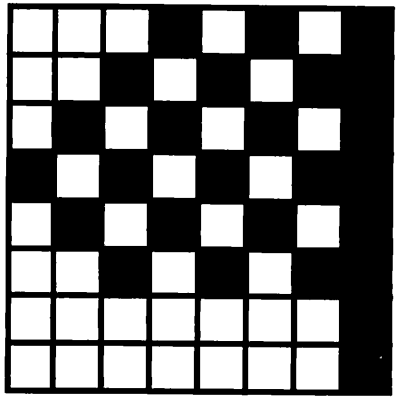
Program 5-1. Setup

```
10 REM MOVE THE NET
20 CALL CLEAR
30 CALL SCREEN(2)
40 CALL COLOR(2,5,1)
50 CALL CHAR(42,"152B55AB552B0101")
60 LET NOR=23
65 LET NNR=23
70 LET NOC=16
75 LET NNC=16
```

This looks similar to the beginning of the program from the last chapter that set up the butterfly. CALL CLEAR and CALL SCREEN are the same, but CALL COLOR has a 2 for the first value because you'll be using a different character set. Since you're creating a net to catch the butterfly, it would be nice to have it a different color. If you used the same character set as the butterfly, it would have to be the same color. In this example, character number 42, which is the asterisk, will be used from set 2.

In line 50, the asterisk character is redefined as a butterfly net. Here is what it would look like:

Figure 5-1. Butterfly Net



The code values used to create the butterfly net are:

152B55AB552B0101

In lines 60 and 70, the variables NOR and NOC stand for the *N*et *O*ld *R*ow and the *N*et *O*ld *C*olumn. They're set to 12 and 16 to put the net in the center of the screen. Lines 65 and

75 set up an initial value of the *Net New Row* and the *Net New Column*, variables NNR and NNC. You'll want to have the new and old row and columns be the same before you actually move.

Now you're ready to put it on the screen. The next line does this:

```
80 CALL VCHAR(NOR,NOC,42)
```

The Main Loop. Now you're ready to set up your main loop. This is usually a very short part of the program that will just go through and do one of two things:

- The program will check to see if a key is pressed, or
- Some form of automatic motion will take place, while the player is deciding what to do.

In this example, only the first thing, which is to see which keys are pressed, will take place.

Program 5-2. Main Loop

```
100 REM  MAIN LOOP
110 CALL KEY(3,K,S)
120 IF K=69 THEN 300
130 IF K=88 THEN 400
140 IF K=83 THEN 500
150 IF K=68 THEN 600
190 GOTO 100
```

Lines 100 and 190 make up the main loop. The program will go back and forth many times each second, and it will do only what is between lines 100 and 190.

In this case, line 110 uses the CALL KEY command. 3 is the key-unit and tells you that the keyboard will be used in an all-uppercase way. *K* is the return variable, and tells you which key, if any, has been pressed. *S* is the status-variable, and won't be used in this example, but since you must have something in its place, we'll use *S*.

Lines 120-150 check to see if the key has been pressed. If it has, you are interested only in whether the keys E, X, S, or D were pressed. For example, if the E key was pressed, line 120 will tell the program to go to line 300. Similarly, the other lines tell the program to go to specific lines, depending on which key is pressed. If none of the four was pressed, the loop continues.

Which Way to Move. Enter these lines for the actual movement subroutines:

Program 5-3. Movement Subroutines

```
300 REM  E MOVE UP
310 NNR=NOR-1
320 GOTO 700
400 REM  X MOVE DOWN
410 NNR=NOR+1
420 GOTO 700
500 REM  S MOVE LEFT
510 NNC=NOC-1
520 GOTO 700
600 REM  D MOVE RIGHT
610 NNC=NOC+1
620 GOTO 700
```

These lines do the arithmetic calculations that will be used to move the net to its new location. NNR and NNC are variables for the *N*et *N*ew *R*ow and the *N*et *N*ew *C*olumn. To move the net up, you simply subtract 1 from the old row. The subroutine from lines 300-320 does this. If you want it to move down, you add 1 to the old row. Lines 400-420 move the net down. The columns change the same way, with subroutines moving the character left in lines 500-520 and right on lines 600-620.

A move changes the row or column by one, either adding or subtracting. Notice that lines 300, 400, 500, 600 all are REM statements. A careful use of REM statements is a good programming habit because you'll want to make sure you're doing the right things in the right places. Typing REM E MOVE UP will remind you that the E key moves things up.

Notice that you have not actually moved anything yet. The new row and column are calculated, but the program must do one more thing before it can move the net.

Checking for Edges. Whenever you have a character moving, you need to check to see that the player's move is legal. Illegal moves can cause the program to stop, and you may not even want the player-controlled figure moving to certain places on the screen. To check for the screen edges with our net movement program, you need to add only these lines:

Program 5-4. Checking Edges

```
700 REM  CHECK MOVE
710 IF (NNR<1)+(NNR>24)THEN 800
720 IF (NNC<1)+(NNC>32)THEN 800
730 CALL VCHAR(NOR,NOC,32)
740 CALL VCHAR(NNR,NNC,42)
```

```
750 NOR=NNR
760 NOC=NNC
770 GOTO 100
```

Lines 700-770 check to see if an error has occurred, and if it has, the program goes to line 800. Since the screen row and column must be between rows 1 and 24 and columns 1 and 32, these are the legal boundaries. If the new row or column would be outside these, an error has occurred.

If there's been no error, the old position of the net is erased and its new position is put on the screen. Finally, lines 750 and 760 change the new row and column into the old row and column so that the program will be ready for the next move. After this has happened, the program goes back to line 100, to wait for the next key to be pressed.

Running into the Edges. All that's left to do is to decide what should happen when an error is committed. In this example, nothing will happen; the character seems to wait patiently at any screen edge.

Program 5-5. Staying Still

```
800 REM  ERROR
810 NNR=NOR
820 NNC=NOC
830 GOTO 100
```

Nothing happens here because the new net will not be printed. This subroutine cancels the calculations of lines 300-620 by resetting the variables NNR and NNC to what they were before. You must always, if you want *not* to do something, make sure that the steps that led up to that decision don't affect other parts. If, for example, you don't change the new row and column to the old row and column when the program discovers an error, the next time you check NNR or NNC, the program will think they are still out of bounds, because you didn't reset them.

You'll find that the following steps will be repeated for most game programs.

- Set up your initial variables, clear the screen, choose you colors, create your character shape, and put your character on the screen.
- Create a main loop that looks for keys pressed or creates other kinds of automatic motion.
- Create smaller parts of the program, called subroutines, that do specific things. In this example, subroutines

were created to calculate new rows and columns for the net, to check for errors and screen edges, as well as to force the character to remain on the screen.

- If there was no error, the old character was erased, a new one put on the screen, and the variables for new row and column were reset to the old row and column, so that the main loop can execute again.

The complete program to create keyboard-controlled movement would look like this:

Program 5-6. Keyboard Movement

```
10 REM  MOVE THE NET
20 CALL CLEAR
30 CALL SCREEN(2)
40 CALL COLOR(2,3,1)
50 CALL CHAR(42,"152B55AB552B0101")
60 NOR=12
65 NNR=12
70 NOC=16
75 NNC=16
80 CALL VCHAR(NOR,NOC,42)
100 REM  MAIN LOOP
110 CALL KEY(3,K,S)
120 IF K=69 THEN 300
130 IF K=88 THEN 400
140 IF K=83 THEN 500
150 IF K=68 THEN 600
190 GOTO 100
300 REM  E MOVE UP
310 NNR=NOR-1
320 GOTO 700
400 REM  X MOVE DOWN
410 NNR=NOR+1
420 GOTO 700
500 REM  S MOVE LEFT
510 NNC=NOC-1
520 GOTO 700
600 REM  D MOVE RIGHT
610 NNC=NOC+1
620 GOTO 700
700 REM  CHECK MOVE
710 IF (NNR<1)+(NNR>24)THEN 800
720 IF (NNC<1)+(NNC>32)THEN 800
730 CALL VCHAR(NOR,NOC,32)
740 CALL VCHAR(NNR,NNC,42)
750 NOR=NNR
760 NOC=NNC
```

```
770 GOTO 100
800 REM  ERROR
810 NNR=NOR
820 NNC=NOC
830 GOTO 100
```

Joysticks

If you have a joystick for your TI computer, you can use it instead of pressing a key. The TI is set up to use two joysticks that are wired together. For this example, only one joystick will be used.

The joystick uses the CALL JOYST command, and needs three steps to move a figure on the screen.

Key-unit. The first part of the CALL JOYST command is called the *key-unit*. This simply tells the computer which joystick you want to test. A key-unit of 1 tests joystick 1, while the key-unit of 2 tests joystick 2.

For most of these examples, joystick 1 will be used.

Joystick Directions. The CALL JOYST command needs two variables, which will hold the column and row *directions* that the joystick pushes toward. If you push it to the right, it gives a number which can be used to calculate a column to the right. Similarly, pushing the joystick to the left gives a number that can be used to calculate a column to the left. The same procedure also applies to the row direction calculations when the joystick is pushed up or down.

Here is the form for CALL JOYST:

CALL JOYST (*key-unit, column-direction, row-direction*)

Notice that the row and column are the opposite of what you might expect from using the CALL VCHAR commands. This is because TI calls them *x-return* and *y-return* and uses *x* to represent the column number and *y* for the row number. Since row and column will be more important here, the words column-direction and row-direction will be used.

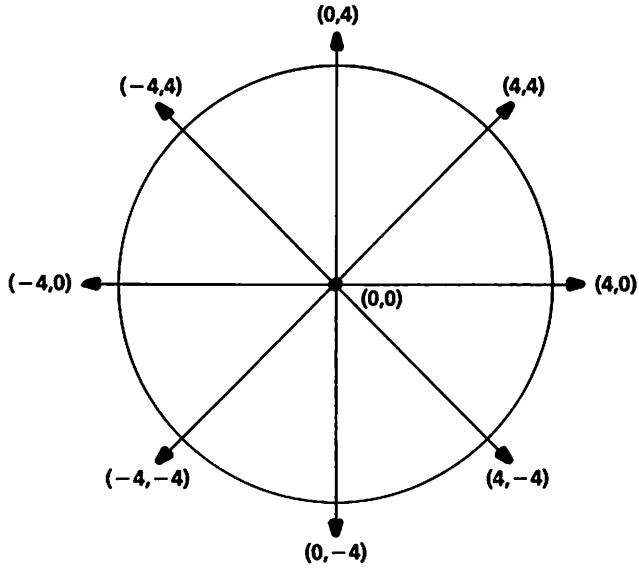
Column-direction and row-direction are variables that you can choose. Each variable, when the CALL JOYST is used, will store certain numbers. Here they are:

Joystick Direction	Column Value	Row Value
Up	0	4
Down	0	−4
Left	−4	0

Joystick Direction	Column Value	Row Value
Right	4	0
Up-Left	−4	4
Up-Right	4	4
Down-Left	−4	−4
Down-Right	4	−4

Figure 5-2 shows these values for the joystick as if you were looking from above.

Figure 5-2. Joystick Direction Values



If you pull the joystick up and to the right, say, the row and column direction variables will be 4 and 4.

(Note: When using the joystick, make sure that the ALPHA LOCK key is up. Otherwise, the joystick up pull will be ignored.)

You can use these variable values in much the same way that you used CALL KEY to control motion.

Example of CALL JOYST. In Program 5-6 in which you just saw how to use CALL KEY, you could use CALL JOYST instead. All you have to do is type these lines:

Program 5-7. Joystick Main Loop

```
110 CALL JOYST(1,JC,JR)
120 IF JR=4 THEN 300
130 IF JR=-4 THEN 400
140 IF JC=-4 THEN 500
150 IF JC=4 THEN 600
```

These will erase the old lines 110-150 and create similar ones that use CALL JOYST.

In this example, you'll use joystick 1. JC is the column variable that stores the column-direction value, and JR will be the row variable used to store the row-direction value.

Lines 120-150 send the program to the appropriate subroutines that will move the net character in the direction the joystick is moved. For example, if JR is equal to 4, the joystick was moved up.

Diagonal Directions. What happens if you pull the joystick in a diagonal direction? The way the program is written now, the up and down movements of a diagonal push on the joystick will happen first, because they're tested first in the main loop. If you want to test for true diagonal movement, you can add these lines:

Program 5-8. Diagonal Joystick Movement

```
305 IF JC<>0 THEN 100
405 IF JC<>0 THEN 100
505 IF JR<>0 THEN 100
605 IF JR<>0 THEN 100
```

This will make sure that the character will move only if the joystick is pushed straight up. By using <> (SHIFT COMMA and SHIFT PERIOD), you are saying that JC or JR is not equal to 0. If it is not 0, you are in a diagonal pull, and the program goes back to line 100.

Because the TI joystick is somewhat unresponsive, you may or may not want to use lines 305, 405, 505, and 605 to keep your directions equal. If you don't, the up and down directions will be used more often, unless you change the order of lines 120-150. Most of the examples in this book will use something like lines 305, 405, 505, and 605 so that the directional movements are equal to the up, down, right, and left directions.

To use a joystick on the TI to move the net figure, the complete program would look like this:

Program 5-9. Joystick Movement

```
10 REM  MOVE THE NET
20 CALL CLEAR
30 CALL SCREEN(2)
40 CALL COLOR(2,3,1)
50 CALL CHAR(42,"152B55AB552B0101")
60 NOR=12
65 NNR=12
70 NOC=16
75 NNC=16
80 CALL VCHAR(NOR,NOC,42)
100 REM  MAIN LOOP
110 CALL JOYST(1,JC,JR)
120 IF JR=4 THEN 300
130 IF JR=-4 THEN 400
140 IF JC=-4 THEN 500
150 IF JC=4 THEN 600
190 GOTO 100
300 REM  MOVE UP
305 IF JC<>0 THEN 100
310 NNR=NOR-1
320 GOTO 700
400 REM  MOVE DOWN
405 IF JC<>0 THEN 100
410 NNR=NOR+1
420 GOTO 700
500 REM  MOVE LEFT
505 IF JR<>0 THEN 100
510 NNC=NOC-1
520 GOTO 700
600 REM  MOVE RIGHT
605 IF JR<>0 THEN 100
610 NNC=NOC+1
620 GOTO 700
700 REM  CHECK MOVE
710 IF (NNR<1)+(NNR>24)THEN 800
720 IF (NNC<1)+(NNC>32)THEN 800
730 CALL VCHAR(NOR,NOC,32)
740 CALL VCHAR(NNR,NNC,42)
750 NOR=NNR
760 NOC=NNC
770 GOTO 100
800 REM  ERROR
810 NNR=NOR
820 NNC=NOC
830 GOTO 100
```

Keyboard and Joystick Control 5

Making a Game

Putting the Butterfly in. To make the butterfly and net programs into a game, all you have to do is add the butterfly.

Since you already have a net, and you created a butterfly in Chapter 4, it's a simple thing to mix the butterfly program parts into the net program. Then, all you have to do is test to see if the net and the butterfly are at the same place at the same time.

In this game, your score will depend on how fast you catch the butterfly with the net. To make it more difficult, the butterfly will start out at a random place on the top of the screen, and your net will start out at the bottom center.

Starting Out. First you must add the butterfly:

Program 5-10. Adding the Butterfly

```
45 CALL COLOR(1,7,1)
55 CALL CHAR(33,"2499DBFF7EFFC381")
90 LET OR=INT(RND*4)+1
95 LET OC=INT(RND*32)+1
97 CALL VCHAR(OR,OC,33)
99 LET COUNT=0
```

These lines create the same butterfly as in Chapter 4. Lines 90 and 95 put it on the screen between row 1 and row 4 and between column 1 and column 32.

Line 99 starts the COUNT variable at 0, and this is used to keep track of the score. Most games are better if you can think of a way to keep score. In this game, the *longer* you play, the higher the *number* of your score. However, the idea here is to get as *low* a score as possible.

Type these two lines to change the beginning location of the net:

```
60 LET NOR=23
65 LET NNR=23
```

The Main Loop. Now add this line to the main loop:

```
102 LET COUNT=COUNT+1
```

This will add 1 to the variable COUNT, so that it will get larger each time the main loop executes. The number of times through will equal your score. The lower the number, the better your score will be.

Enter:

```
105 GOSUB 200
```

This will send the program to line 200, which is a subroutine to print the butterfly. Since the loop always goes back to line 100, you want to make sure you move the butterfly before you check to see if you move the net. If you do it the other way, the butterfly will never get a chance to move.

Move Butterfly Subroutine. Here's the subroutine which will move the butterfly:

Program 5-11. Butterfly Movement Subroutine

```
200 REM MOVE BUTTERFLY
210 I=INT(RND*3)-1
220 J=INT(RND*3)-1
230 LET NR=OR+I
240 LET NC=OC+J
250 IF NR<1 THEN 900
255 IF NR>24 THEN 900
260 IF NC<1 THEN 900
265 IF NC>32 THEN 900
270 CALL VCHAR(OR,OC,32)
275 CALL VCHAR(NR,NC,33)
280 LET OR=NR
285 LET OC=NC
290 IF NR=NOR THEN 1000
292 IF NC=NOC THEN 1100
295 RETURN
```

Lines 200-285 are the same as those used in the last chapter to create and move the butterfly.

Lines 210-240 create a new row and column by using random numbers. Lines 250-265 check the new row and column to see if there are any errors. If there are, the program goes to line 900. Otherwise, the program continues, and lines 270 and 275 erase the old butterfly and put the new one on the screen.

Lines 280 and 285 make the new row and column into the old row and column so that the next time through, the butterfly will be ready to move.

Lines 290 and 292 are new: They check to see if a collision has occurred between the butterfly's old row and column, and the net's old row and column. If the rows match (NR = NOR), the program goes to line 1000. If the columns match (NC = NOC), the program goes to line 1100.

Line 295 RETURNs the program to the main loop.

Butterfly Errors. Type these lines to take care of any error that occurs if the butterfly flies too close to the edge:

Program 5-12. Butterfly Edges

```
900 REM BUTTERFLY ERROR
910 LET NR=OR
920 LET NC=OC
930 GOTO 100
```

These lines set the new row and column back to their old values, so that nothing happens when the butterfly tries to fly off the screen. In other words, it will always remain on the screen.

Checking for Collisions. You would add these lines to check for the collisions between the butterfly and the net.

Program 5-13. Collision Checking

```
1000 REM ROW MATCH, CHECK COLUMN
1010 IF NC=NOC THEN 2000
1020 RETURN
1100 REM COLUMN MATCH, CHECK ROW
1110 IF NR=NOR THEN 2000
1120 RETURN
```

Lines 1000-1020 were accessed from line 290, which found a match between the *rows* of the butterfly and net. If a match is found in line 1010 between the columns of the butterfly and the net, you know that they must both be in the same row and column. If this is so, the program will go to line 2000, which is the end. If it is not a match—that is, if only the rows match—the program RETURNs to the main loop.

Lines 1100-1120 serve the same purpose. Accessed from line 292, which found a match between the *columns* of the butterfly and net, line 1110 will shift the program to the subroutine at line 2000 if NR = NOR. If the columns do not match, the program RETURNs to the main loop.

Ending the Game. If the butterfly and net have collided, the game is over. The short subroutine that follows acts as an end routine for our game.

You can also program your game to let the player choose if he or she wants to play again. This is simple to do, and the end routine includes this feature.

Program 5-14. Endings

```
2000 REM COLLISION
2005 CALL CLEAR
2007 CALL SCREEN(3)
2010 PRINT "YOU CAUGHT THE BUTTERFLY"
```

```
2020 PRINT "IN ";COUNT;" MOVES"
2030 PRINT "WANT TO PLAY AGAIN?"
2040 PRINT "PRESS Y FOR YES"
2050 CALL KEY(3,K,S)
2060 IF S=0 THEN 2050
2070 IF K=89 THEN 10
2080 END
```

This routine ends the game by clearing the screen and changing the screen color. The screen color must be changed because the letters on the screen normally print in black, and since we had a black screen, the letters wouldn't show up unless you changed the screen color.

Lines 2010-2040 let the player know that the game is over, and what the score was, by printing the COUNT in line 2020.

Notice the way that line 2020 is spaced, and also the use of the semicolon, to allow text and variables to be mixed in the same print line.

CALL KEY is used in line 2050 to let the player choose to play again. S is the status-variable, and if S is 0, no key has been pressed. If none was pressed, the program goes back to line 2050 and keeps going back until a key is pressed, which changes the S variable to something else besides 0.

If a *Y* was pressed, K is set to 89, the ASCII number for *Y*. If *Y* was pressed, the program begins again at line 10. If the player typed anything other than a *Y*, the game is over.

Your First Game

"Flutters," the butterfly and net game you've been developing as you read, is your first complete game. Although you have seen it in bits and pieces, it may be worthwhile to see it in a complete form. Here it is.

Program 5-15. Flutters

```
10 REM  MOVE THE NET
20 CALL CLEAR
30 CALL SCREEN(2)
40 CALL COLOR(2,5,1)
45 CALL COLOR(1,7,1)
50 CALL CHAR(42,"152B55AB552B0101")
55 CALL CHAR(33,"2499DBFF7EFFC381")
60 LET NOR=23
65 LET NNR=23
70 LET NOC=16
75 LET NNC=16
80 CALL VCHAR(NOR,NOC,42)
```

```
90 LET OR=INT(RND*4)+1
95 LET OC=INT(RND*32)+1
97 CALL VCHAR(OR,OC,33)
99 LET COUNT=0
100 REM  LOOP
102 LET COUNT=COUNT+1
105 GOSUB 200
110 CALL JOYST(1,JC,JR)
120 IF JR=4 THEN 300
130 IF JR=-4 THEN 400
140 IF JC=-4 THEN 500
150 IF JC=4 THEN 600
160 CALL SOUND(NOR*10+10,NOC*100+200,0)
190 GOTO 100
200 REM  MOVE BUTTERFLY
210 I=INT(RND*3)-1
220 J=INT(RND*3)-1
230 LET NR=OR+I
240 LET NC=OC+J
250 IF NR<1 THEN 900
255 IF NR>24 THEN 900
260 IF NC<1 THEN 900
265 IF NC>32 THEN 900
270 CALL VCHAR(OR,OC,32)
275 CALL VCHAR(NR,NC,33)
280 LET OR=NR
285 LET OC=NC
290 IF NR=NOR THEN 1000
292 IF NC=NOC THEN 1100
295 RETURN
300 REM  MOVE UP
305 IF JC<>0 THEN 100
310 LET NNR=NOR-1
320 GOTO 700
400 REM  MOVE DOWN
405 IF JC<>0 THEN 100
410 LET NNR=NOR+1
420 GOTO 700
500 REM  MOVE LEFT
505 IF JR<>0 THEN 100
510 LET NNC=NOC-1
520 GOTO 700
600 REM  MOVE RIGHT
605 IF JR<>0 THEN 100
610 LET NNC=NOC+1
620 GOTO 700
700 REM  CHECK MOVE
710 IF (NNR<1)+(NNR>24)THEN 800
720 IF (NNC<1)+(NNC>32)THEN 800
745 CALL VCHAR(NOR,NOC,32)
```

```
750 CALL VCHAR(NNR,NNC,42)
760 LET NOR=NNR
770 LET NOC=NNC
780 GOTO 100
800 REM  ERROR
810 LET NNR=NOR
820 LET NNC=NOC
825 CALL SOUND(100,-6,0)
830 GOTO 100
900 REM  BUTTERFLY ERROR
910 LET NR=OR
920 LET NC=OC
930 GOTO 100
1000 REM  ROW MATCH, CHECK COLUMN
1010 IF NC=NOC THEN 2000
1020 RETURN
1100 REM  COLUMN MATCH, CHECK ROW
1110 IF NR=NOR THEN 2000
1120 RETURN
2000 REM  COLLISION
2005 CALL CLEAR
2007 CALL SCREEN(3)
2010 PRINT "YOU CAUGHT THE BUTTERFLY"
2020 PRINT "IN ";COUNT;" MOVES"
2030 PRINT "WANT TO PLAY AGAIN?"
2040 PRINT "PRESS Y FOR YES"
2050 CALL KEY(3,K,S)
2060 IF S=0 THEN 2050
2070 IF K=89 THEN 10
2080 END
```

Saving Your Game. When you've finished typing all of this in, type RUN and press ENTER. You must catch the butterfly as quickly as you can. Each time you do, you'll be told how long it took you, and you can try to improve your score.

Before you turn off your computer, however, be sure to save a copy of your game onto your cassette recorder or disk drive. Typing SAVE CS1 if you have a cassette recorder, or SAVE DSK1.FLUTTER if you have a disk drive, will place the game under the filename FLUTTER.

Changing the Game

There are many things you could do to change and improve this game.

- You could make the butterfly fly farther each time by making it add or subtract a number larger than 1 to its row or column.

- You could put in more than one butterfly.
- You could have butterflies flying in a pattern from left to right, and try to see how many you could catch in a net during a given time.

Review

In this chapter you've seen how to use the keyboard and the joystick to move an object around on the screen.

CALL KEY, which is used to detect a pressed key, has three parts:

- The *key-unit,* which is used to define how the keyboard will be read. Usually key-unit 3 is used.
- The *return-variable* stores the ASCII number of the pressed key so that you can use it later in the program.
- The *status-variable* stores the status of the key that was pressed. If the status-variable equals 1, a new key was pressed. If it equals −1, the same key was pressed, and if it is equal to 0, no key was pressed.

The form for CALL KEY is:

CALL KEY, *(key-unit, return-variable, status-variable)*

CALL JOYST reads the joysticks and needs three things:

- The *key-unit,* which will be 1 for joystick 1, or 2 for joystick 2.
- The *x-return,* which will be the same as the column direction, left or right. The numbers stored will be 0, 4, or −4.
- The *y-return,* which will be the same as the row direction, up or down. The numbers stored will be 0, 4, or −4.

The form for CALL JOYST is:

CALL JOYST *(key-unit, left-right, up-down)*

Tools of the Trade

You have all the BASIC building blocks that you'll need to create your own arcade games. As you have seen in this chapter, it's relatively easy to design and write individual pieces of a program, which, when added together, create an entire game. In fact, you created a game that nets butterflies. By using what you've already learned, you can create all kinds of arcade games.

But you can make your game program even better by adding sound to it. Sound effects can enhance any game, and we almost always expect them in an arcade-style game. Chapter 6 introduces you to the TI's sound capability.

6 Sound

6 Sound

Using the CALL SOUND Command

You can enhance your games by using sound to provide information to the player, as well as making the game more entertaining. The command CALL SOUND allows you to create music, noises, or even individual tones.

CALL SOUND needs at least three numbers to create a sound:

- *Duration*, which tells the computer how long to make the sound. You can use a number from 1 to 4250, each increment representing 1/1000 second. If you use 400, for example, you'll get the sound's duration for 4/10 second; 4000 would set the duration to 4 seconds.
- *Frequency*, or the pitch of the sound. If it is a positive number, the range can go from 110 to 44733. This is expressed as cycles per second, with 110 being a very low tone and 44733 higher than the human ear can hear. The *User's Reference Guide*, your manual to the TI, includes an appendix listing the numbers which produce various pitches on the musical scale. For example, 262 represents a middle C on the musical scale. If you know how to read music, you can make your TI play tunes. You can play up to three musical notes at one time, so you can create three-part harmony.

If the number is negative, it must be between −1 and −8. Negative numbers produce periodic and white noises, which can be used for special effects.

Here's how negative numbers work for the frequency part of CALL SOUND:

Table 6-1. Noise Frequency Values

Number	Effect
–1	High pitched buzz
–2	Medium pitched buzz
–3	Low pitched buzz
–4	A buzz related to the third tone in the group of tones.
–5	High pitched white noise
–6	Medium pitched white noise
–7	Low pitched white noise

–8 White noise that is related to the third tone in the group of tones.

You can use buzzes for electronic sounds, while white noises can be used for wind, explosions, surf, static, and so on.

- *Volume*, which is how loud a sound will be. 0 is the loudest, and 30 the softest.

You can use CALL SOUND to create up to three sounds and one noise at once by adding more frequency and volume numbers after the initial set.

CALL SOUND looks like this:

CALL SOUND *(duration, frequency, volume)*

You can add more frequencies and volumes after the first three variables, but be sure to always add them in pairs, frequency first, followed by volume. You can have four sounds in a single CALL SOUND command, but only one of them can be a noise value (−1 to −8). If you have more than one sound in a single CALL SOUND command, they'll play at the same time.

Here's a typical sound:

```
CALL SOUND (100,262,0)
```

This would produce a middle C (262 cycles per second), last 1/10 second, at the loudest volume of 0.

If you wanted to add a second note, you could change the command to:

```
CALL SOUND (100,262,0,294,0)
```

This would add a middle D sound, also at the loudest volume, to play two notes at once.

Sounds in Your Program. When you use CALL SOUND, the TI will process the sound commands independently of the rest of the program. In other words, the program will continue with whatever comes after the CALL SOUND command, especially if the sound's duration is long. You can use this to play music or sound effects while the program is running, but when there is no motion on the screen. Because of this, you must plan your sounds so that they'll end when you want them to. Often a sound will continue while other things take place, so you must try to time them by estimating how long you want the sound to continue after the program has processed the CALL SOUND command.

You can also use CALL SOUND to provide a constant background sound as the game is played. Let's add sound to "Flutters," the game we created in Chapter 5.

```
160 CALL SOUND(NOR*10+10,NOC*100+200,0)
```

Adding this line will create different sounds, depending on where you move the net.

The variable NOR is the row and is used to calculate the duration of the sound. If NOR is small, the duration will be short.

NOC is the variable for the column and is used to calculate the frequency. The smaller the value for NOC, the lower the frequency.

When you add this line, you can tell where the net is by how long the tone plays and what its pitch is.

You can add this line to create a sound effect in Flutters:

```
825 CALL SOUND(100,-6,0)
```

Now you'll hear a short crashing sound whenever the net bumps into the screen edge. The duration is 1/10 second, and the tone is one of the white noise sounds the TI can make.

By using this, you can always tell when you've hit the edge.

Sounds can be very useful for telling the player what has happened, providing player feedback, as well as dressing up the game by playing music.

Sound Parade

To see how to create several useful sounds, here's a program demonstrating effects you can use in your *own* games:

Program 6-1. Sound Parade

```
10 REM SOUND PROGRAM
20 CALL CLEAR
30 CALL SCREEN(7)
40 RANDOMIZE
50 GOSUB 2000
60 LET Z$="123456789ABC"
65 CALL KEY(3,K,S)
70 IF S<1 THEN 65
75 LET F$=CHR$(K)
80 FOR I=1 TO 12
83 IF F$=SEG$(Z$,I,1)THEN 90
85 NEXT I
87 GOTO 65
```

6 Sound

```
90 LET X=I
100 ON X GOSUB 200,300,400,500,600,700,800,900,100
    0,1100,1200,1300
110 GOTO 65
200 REM SCALES
205 RESTORE
210 FOR I=1 TO 8
220 READ A
230 CALL SOUND(200,A,0)
240 NEXT I
250 DATA 262,294,330,349,392,440,494,523
290 RETURN
300 REM TUNE 3 PART HARMONY
310 RESTORE
320 FOR I=1 TO 8
325 READ A
326 LET B(I)=A
330 NEXT I
340 FOR I=1 TO 30
350 LET R=B(INT(RND*8)+1)
352 LET S=B(INT(RND*8)+1)
354 LET T=B(INT(RND*8)+1)
360 CALL SOUND(100,R,0,S,0,T,0)
370 NEXT I
390 RETURN
400 REM FALLING SOUND
410 FOR I=1 TO 30
420 CALL SOUND(100,2000-50*I,0)
425 NEXT I
490 RETURN
500 REM BOUNCE
510 CALL SOUND(100,110,0)
530 CALL SOUND(100,440,0)
590 RETURN
600 REM WAVES
610 FOR I=1 TO 4
620 CALL SOUND(400+INT(RND*200),-7,0)
622 CALL SOUND(2000,-6,5)
627 CALL SOUND(4000,-5,12)
630 CALL SOUND(INT(RND*200),44000,15)
650 NEXT I
690 RETURN
700 REM TICK TOCK
705 FOR J=1 TO 8
710 CALL SOUND(10,-5,0)
720 GOSUB 3000
740 CALL SOUND(10,-7,0)
745 GOSUB 3000
750 NEXT J
```

```
790 RETURN
800 REM ALERT
810 FOR I=1 TO 8
820 CALL SOUND(200,440,0,400,0)
830 CALL SOUND(200,880,0,800,0)
840 NEXT I
890 RETURN
900 REM URGENT RUNNING
910 FOR I=1 TO 30
920 CALL SOUND(10,440,0,450,0)
930 CALL SOUND(10,450,0,460,0)
940 CALL SOUND(10,460,0,470,0)
950 NEXT I
990 RETURN
1000 REM UFO MOVING
1004 FOR J=1 TO 8
1005 FOR I=1 TO 7
1010 CALL SOUND(100,-1,I,2000,0)
1070 NEXT I
1080 NEXT J
1090 RETURN
1100 REM DEATH RAY
1105 FOR I=1 TO 8
1110 CALL SOUND(100,-3,0,880,0,890,0)
1150 NEXT I
1190 RETURN
1200 REM CRASH
1205 CALL SOUND(200,-5,0)
1210 FOR I=1 TO 3
1220 CALL SOUND(200*I,-7,I*5)
1230 NEXT I
1290 RETURN
1300 REM BLAST OFF
1305 FOR I=1 TO 15
1310 CALL SOUND(50,-5,4)
1315 CALL SOUND(10,900+30*I,0)
1320 NEXT I
1350 CALL SOUND(2000,-7,0)
1390 RETURN
2000 REM TITLES
2005 PRINT "LIBRARY OF SOUNDS"
2006 PRINT
2010 PRINT "TO HEAR A SOUND"
2015 PRINT "PRESS THE APPROPRIATE KEY"
2017 PRINT "(ALPHA LOCK DOWN)"
2020 PRINT
2030 PRINT "1  - SCALES"
2040 PRINT "2  - TUNE 3 PART HARMONY"
2050 PRINT "3  - FALLING SOUND"
```

6 Sound

```
2060 PRINT "4  - BOUNCE"
2070 PRINT "5  - WAVES"
2080 PRINT "6  - TICK TOCK"
2090 PRINT "7  - ALERT"
2100 PRINT "8  - URGENT RUNNING"
2110 PRINT "9  - UFO MOVING"
2120 PRINT "A  - DEATH RAY"
2130 PRINT "B  - CRASH"
2140 PRINT "C  - BLAST OFF"
2200 FOR I=1 TO 4
2210 PRINT
2220 NEXT I
2900 RETURN
3000 REM DELAY 1
3010 FOR DELAY=1 TO 200
3020 NEXT DELAY
3030 RETURN
```

When you type in the program and RUN it, you'll see several choices displayed. Press the appropriate button to hear a specific sound.

Each of the subroutines between lines 200 and 1390 contains a set of BASIC commands that use CALL SOUND in various ways to produce a sound or group of sounds. Each of the sound subroutines is called by the ON . . . GOSUB command in line 100. You can analyze each of the subroutines and see how they work, and even insert them into your own game program.

You can look through each of the subroutines and see how a slight change in one of the CALL SOUND variables will alter the sound when it plays. By experimenting, you'll be able to come up with your own sounds.

Using Sound

Sounds can be used in a game program to do several things. You can play music at the beginning of a program. If your program will take a long time to set up, a tune can keep the player's interest while your program creates the screen and sets variables.

You can add sounds for each action that happens, so that you create a lively game. Arcade-game players like to have sounds with their games, to add entertainment and interest.

However, don't think that sounds are just for entertainment. Sounds can be very important. You can use sounds to give the

player information. You can use sounds to alert the player that something new has happened; for example, a new wave of opponents has started down the screen, or the clock shows less time. When your player-controlled figure crashes or explodes, you can reinforce the loss by sound, and if the player wins, you can use sound to reinforce the victory.

Review

CALL SOUND. To add sound to your program you can use CALL SOUND, which includes three necessary values:

- *Duration*. The values range from 1 to 4250 and are measured in 1/1000 second.
- *Frequency*. Values range from 110 to 44733 and are measured in cycles per second.
- *Volume*. 0 is loudest and 30 is softest.

The form for CALL SOUND is:

CALL SOUND *(duration, frequency, volume)*

Creating Your Game

In the chapters which follow, you'll see many different kinds of games, but all of them will have the same basic structure.

Here are the three parts of my arcade game structure:

- The setup—determining the initial values for the objects on the screen.
- The main loop—checking to see if a key is pressed or a joystick has been pushed, or moving a computer-controlled figure, such as the butterfly in Flutters.
- Smaller programs—some of these are subroutines, and others are just small program pieces that are executed by GOTO statements. It helps to break a program down into parts that you can easily deal with and change. Often you can reuse these small programs and subroutines in other programs.

Analyzing other programmers' games is almost as educational as writing your own. When you look over a game program listing, you can often find techniques you'll want to use.

As you go through each of the following game chapters, you'll not only have games that you can type in and play, you'll also see how different kinds of arcade games are programmed. You'll see hints on how to modify the games by

making simple changes and in most cases see the program lines to add or alter to make one of the modifications. By using the principles that you've learned in the first six chapters, as well as both old and new techniques in the games themselves, you'll be able to write your own arcade games on the TI computer.

The last game chapter focuses on the use of sprites and other features which you can use if you have the TI Extended BASIC cartridge. Sprites are special characters that can move on the screen all by themselves, and have other properties that make for even more exciting arcade games.

A final chapter will give you a step-by-step process that you can use to design your own games.

7 Martian Attack

7 Martian Attack

String arrays can be used to store information, in this case the computer-controlled opponents, in a game. You'll also see how to use the joystick to operate the player-controlled character.

How to Play

After you've entered this program, type RUN and press ENTER.

When you do, there'll be a short pause, and the screen will go black. You'll see a purple cannon at the bottom of the screen, and four orange attackers at the top.

They'll move down at a constant speed, making beeping noises as they go. But the invaders keep shifting their position, making it hard to stop them before they reach ground level. You fire your cannon by pressing the fire button on joystick number 1. A green missile will shoot up and eliminate an invader—if your aim was good. To move your cannon, push your joystick to the right or left.

If you hit an invader, there will be a short explosion and one of the invaders will disappear. You must get all four invaders before they reach ground level or you'll lose the game.

Destroying all four invaders before they reach ground level ends the game. You'll see a score based on how far away the last invader was when you eliminated it, and the screen will turn red.

Win or lose, you'll have a chance to play again; type *Y* if you want another game.

Program Structure

Like all other games in this book, this one also has three parts. The first part, lines 10-90, is the setup, where the initial values are assigned. The second part is the main loop, which runs from line 100 to 140. The third part, various subroutines, makes up the rest of the program.

Program 7-1. Martian Attack

```
10 REM MARTIAN ATTACK
15 LET GC=16
20 LET GR=23
25 CALL CLEAR
30 CALL SCREEN(1)
```

7 Martian Attack

```
35 CALL COLOR(1,5,1)
40 CALL COLOR(4,13,1)
45 CALL COLOR(2,7,1)
50 CALL CHAR(33,"183C18183C187EFF")
55 CALL CHAR(42,"FFC3663C18181818")
60 CALL CHAR(60,"10383838387C7C44")
65 LET COUNT=0
68 RANDOMIZE
69 FOR I=1 TO 8
70 LET A$(I)=" "
71 NEXT I
72 FOR I=1 TO 4
73 LET A$(I*2)=CHR$(42)
74 NEXT I
75 LET IR=1
80 LET IC=INT(RND*16)+4
90 CALL VCHAR(GR,GC,33)
100 REM LOOP
110 LET COUNT=COUNT+1
113 IF COUNT=10 THEN 200
120 CALL KEY(1,K,S)
125 IF K=18 THEN 300
130 CALL JOYST(1,JC,JR)
135 IF JC=-4 THEN 500
136 IF JC=4 THEN 550
140 GOTO 100
200 REM PRINT INVADERS
206 LET COUNT=0
210 FOR I=1 TO 8
215 CALL VCHAR(IR,IC+I,32)
220 NEXT I
225 LET IR=IR+1
230 IF IR=23 THEN 700
233 LET IC=INT(RND*16)+4
235 FOR I=1 TO 8
240 CALL VCHAR(IR,IC+I,ASC(A$(I)))
242 CALL SOUND(5,800+16*I,0)
245 NEXT I
250 GOTO 120
300 REM FIRE
310 FOR I=22 TO IR+1 STEP -1
320 CALL VCHAR(I,GC,32)
325 CALL VCHAR(I-1,GC,60)
327 CALL SOUND(10,-2,0)
330 NEXT I
335 CALL VCHAR(I,GC,32)
340 FOR I=1 TO 8
345 IF ASC(A$(I))=42 THEN 360
350 NEXT I
```

```
355 GOTO 130
360 IF GC=IC+I THEN 800
370 GOTO 350
500 REM PULL LEFT
510 IF GC=1 THEN 1000
515 CALL SOUND(50,440,0)
520 LET GC=GC-1
530 CALL VCHAR(GR,GC+1,32)
535 CALL VCHAR(GR,GC,33)
540 GOTO 140
550 REM PULL RIGHT
560 IF GC=32 THEN 1000
565 CALL SOUND(50,440,0)
570 LET GC=GC+1
580 CALL VCHAR(GR,GC-1,32)
585 CALL VCHAR(GR,GC,33)
590 GOTO 140
700 REM GAME OVER
710 CALL SCREEN(8)
720 PRINT "THE MARTIANS HAVE LANDED"
730 PRINT "YOU HAVE LOST"
790 GOTO 880
800 REM GET ONE INVADER
810 LET A$(I)=CHR$(32)
815 CALL SOUND(300,-5,0)
820 FOR I=1 TO 8
830 IF ASC(A$(I))=42 THEN 130
840 NEXT I
845 REM END
850 CALL CLEAR
860 CALL SCREEN(10)
865 PRINT "YOU GOT THEM ALL"
870 PRINT "YOUR SCORE WAS ";INT(100*((23-IR)
    /22));"POINTS"
880 PRINT "WANT TO PLAY AGAIN?"
885 PRINT "PRESS Y FOR YES"
887 CALL KEY(3,K,S)
888 IF S=0 THEN 887
889 IF K=89 THEN 10
890 END
1000 REM CRASH
1010 CALL SOUND(100,-6,0)
1020 GOTO 140
```

Here's a line-by-line description of the game program:

Line 10 is the title of the game, put into a REM statement. It's a good idea to have at least the title, and any other information you want to remember about the game, here.

Lines 15-20 initialize the gun column and row at 16 and 23 respectively.

Line 25 clears the screen, and line 30 turns the screen black. Putting these lines at the beginning of the program makes sure the screen begins empty.

Lines 35-45 set the character colors to purple for the exclamation point, green for the < sign, and orange for the asterisk.

Lines 50-60 change the !, <, and * signs to new dot patterns with CALL CHAR.

Line 65 sets the COUNT variable to 0, which is used to decide when to move the invaders down.

Line 68 uses the RANDOMIZE command. If this were not used, the same pattern would repeat every game.

Lines 69-71 contain something you've not yet seen used in this book. Blanks are first put into the string variable array A$. This is where the invaders are stored, and the blanks will remove any display left from an earlier game.

Lines 72-74 put the invaders into the proper locations of A$. Character 42 is the invader; it will be placed into every other location of A$ (I*2).

Lines 75-80 set up the beginning invader row and column. The row (IR) will be 1, at the top of the screen, and the column (IC) will be set by a random number between 4 and 21. Since the invader group is eight columns wide, this will center them.

Line 90 puts the gun (ASCII 33) at its proper row and column on the screen.

Line 100 is a REM to remind you that the main loop starts at line 100. The program will spend most of its time in this loop.

Line 110 adds 1 to the COUNT variable. Line 113 sees if the COUNT variable is equal to 10. If it is, the program jumps to the smaller program at line 200. This will move the invaders down one line.

Line 120 checks the joystick fire button. If it was pressed, line 125 tells the program to go to line 300, which fires a missile.

Line 130 reads the joystick. If it was moved left, line 135 shifts the program to the subroutine at line 500; if it was moved right, line 136 shifts the program to line 550.

Line 140 closes the main loop and sends the program back to line 100.

Line 200 begins the routine that moves the invaders down.

Line 206 changes the COUNT variable back to 0 so that the next time around, the main loop will start counting up from 0 again.

Lines 210-220 PRINT spaces (ASCII 32) to erase the old invaders before PRINTing new ones.

Line 225 increases the row (IR) of the invaders by 1, moving them down.

Line 230 checks to see if the new row of the invaders is equal to 23. If it is, the program jumps to 700, which ends the game.

Line 233 calculates a new column for the invaders between column 4 and column 21.

Lines 235-245 put the new invaders on the screen, by PRINTing the string variable array A$. If some of the invaders have been destroyed, they won't be printed. Line 242 creates a sound as each invader is printed.

Line 250 shifts the program back to the main loop.

Line 300 begins the routine that executes the firing of the missile.

Lines 310-330 fire the missile and produce a sound as it moves. Line 310 calculates where the missile will begin and end. The missile (ASCII 60) is erased in line 320 and put on the screen in line 325. Notice that the missile is erased one row below the place where it will be put on the screen. The missile sound is created in line 327.

Line 335 erases the missile at the last point it was PRINTed, but only after it's moved up as far as possible. This is necessary because of the way that the FOR/NEXT loop works in lines 310-330.

Lines 340-350 check to see if any invaders are in the proper position. If they are, the program jumps to line 360. Otherwise, the FOR/NEXT loop continues, and the program goes back to the main loop at line 130.

Line 360 checks to see if any invader is in the same column as the missile. If it is, the missile has hit one of the invaders, and the routine at line 800 is called. Otherwise, the program uses line 370 to return to the FOR/NEXT loop at line 350 so it can check for more matches.

Lines 500-540 move the player's figure if the joystick was pushed to the left. Line 510 checks to see if the gun row is at the left edge of the screen. If it is, the program jumps to line 1000.

Line 515 produces a sound as the gun moves.

Line 520 decreases the gun column by one, and lines 530 and 535 erase the old gun and print it at its new location.

Line 540 sends the program back to the main loop.

Lines 550-590 operate like lines 500-540, except for right joystick movement.

Lines 700-790 make up the routine which executes when the invaders reach row 23. Line 710 changes the screen color and lines 720-730 print out a message. Then the program goes to line 880, for the ending message.

Lines 800-840 eliminate an invader. Line 810 puts a space into the array A$ so that instead of an invader, a space is PRINTed. An explosion sound effect is produced by line 815, and lines 820-840 check to see if there are invaders left. If there are one or more left, the program returns to the main loop.

Eliminating all the invaders sends the program to the final message starting at line 850.

Line 850 clears the screen, line 860 changes its color, line 865 prints the message, and line 870 calculates your score, based on how low on the screen the invaders reached.

Lines 880-890 let the player choose another game. Line 887 uses CALL KEY to see if a key was pressed; line 888 sees if any key was pressed. If no key was pressed, CALL KEY is used again. If a key was pressed, the program sees, in line 889, if a *Y* (character 89) was pressed. If it was, the program starts again at line 10. If it wasn't, the program ends at line 890 with an END. It's a good programming habit to END your programs to be on the safe side.

Lines 1000-1020 create the crashing sound effect used when the player's figure goes too far to the left or right.

Variables

GC	Gun Column.
GR	Gun Row.
COUNT	Counting variable used to determine when to lower the invaders one row.
A$	String variable array used to store the invaders' characters.
IR	Invader Row.
IC	Invader Column.
K	Key pressed.
S	Key status.

JC	Joystick Column value.
JR	Joystick Row value.

Special Notes. The use of string variable arrays in this example is helpful so that you can keep track of which invaders have been destroyed and which haven't. Notice that you must use a string variable *array* if you want to change what is in the string; otherwise, you could just use a string and read from it by using the SEG$ function.

Arrays can make a program more efficient, especially if a process is repeated many times. The use of a variable array in "Martian Attack" is a good example. For more information on string variable arrays, refer to the TI's *User's Reference Guide* or COMPUTE!'s *Programmer's Reference Guide to the TI-99/4A*.

Changing the Game

There are many ways you could change Martian Attack. Changes of colors, sounds, and character shapes would be simplest.

- If you don't have a joystick, you can use the methods outlined in Chapter 5 to use the keyboard by using CALL KEY.
- You can increase the number of invaders by putting more of them in the array A$ or by making A$ larger and changing the lines that use A$ to reflect this.
- You can make the game faster and harder by letting COUNT go up to a smaller number in line 113, or you can make it slower and easier by making the count go to a higher number.

Although this was designed to be a simple program so that you could analyze the parts easier, you could do much to make this into a more elaborate game. Some of the things you could do are:

- Make it so that each time a group of invaders is destroyed, a new group starts out at a lower position.
- Also, you could design it so that as each new wave begins, a different shape of invader attacks by using CALL CHAR to redefine new characters.
- After going through some of the examples in later chapters, you might want to add a feature that lets the invaders drop missiles on the player's character.
- You could also add other objects that could be shot

down. If you do, you might want to change the scoring to include these objects, or you could change the scoring if you have multiple waves.

- After you have seen later chapters, you could also display the current score, the number of guns left (if you want to add more than one gun destroyed before the game is over), or the high score during multiple game play.

Round 10. For an example of how to actually program a change to the game, you could add a few lines so that you can play ten rounds before the game ends.

Each round of play will have the invaders starting at a lower row than before. You'll get a round score, and then, after ten rounds, you'll see a final score.

It's not difficult. As you make these changes, you'll see how easy it is to modify any arcade-style game.

To begin, you could add these lines to the game program.

```
12 LET ROUND=1
13 LET POINTS=0
```

These lines set the ROUND variable, which is used not only to keep track of which round the game is in, but also to determine where the invaders start on the screen. POINTS is the variable for the total number of points earned.

Now enter these lines:

```
867 LET TEMP=INT(100*((23-IR)/22))
868 LET POINTS=POINTS+TEMP
875 GOTO 3000
```

The temporary variable, TEMP, stores only the points earned that round. TEMP is added to POINTS to create a total score. Line 3000 ends the round. Here's the routine beginning at line 3000:

```
3000 REM  NEXT ROUND
3010 LET ROUND=ROUND+1
3015 FOR I=1 TO 1000
3016 NEXT I
3020 IF ROUND<>11 THEN 15
3030 PRINT "THE GAME IS OVER"
3040 PRINT "YOUR FINAL SCORE IS ";POINTS
3050 GOTO 880
```

These lines increase the ROUND number, create a short delay so you can read the round's message, and then decide whether the program has gone ten rounds. If not, the program

returns to line 15 to start another round.

If ten rounds have elapsed, lines 3030-3040 put the final score on the screen and send the program to line 880 for the end routine.

Some of the program lines need to be changed.

```
75 LET IR=ROUND
```

This line is changed so that IR, the invaders' beginning row, will equal ROUND. As ROUND increases, so will the invaders' row.

```
730 GOTO 867
```

This alters the program logic so that the game will not end here, but go to the end of a round instead.

```
870 PRINT "YOUR ROUND SCORE WAS ";TEMP;" POINTS"
```

The temporary round score, TEMP, is printed instead of the total points.

You'll also have to erase line 790, since line 730 now is a GOTO command.

8 Martian Revenge

8 Martian Revenge

Horizontal scrolling is difficult to simulate on many computors. But by using a large string variable array, you can create a game which makes it appear as if the screen were scrolling from right to left.

How to Play

Your ship starts out at the top, and the city is moving below you. You can drop bombs by pressing the joystick fire button.

If you push the joystick to the right, your ship will go forward, and the city will flow beneath you faster. If you push the joystick to the left, your ship will drop back and the city will scroll under you more slowly.

As you pass over a missile base, it may fire at you. If it does, it may hit you and the game ends.

Hitting a building with a bomb increases your score by 10 points. If you hit a missile base, you receive 25 points. No points are awarded if you hit an empty area.

When the game begins, there will be a pause while the city array is filled, and then the title will PRINT. You see how many bombs you have available and what your score is. You start out with only 20 bombs; each time you drop one, the display changes.

When you crash, the game is over, and your score is shown.

Program 8-1. Martian Revenge

```
10 REM  MARTIAN REVENGE
20 CALL CLEAR
25 CALL COLOR(3,16,1)
26 CALL COLOR(4,16,1)
30 CALL CHAR(93,"0038FF99FFB5FFFF")
31 CALL CHAR(94,"FFABFFD5FFABFFFF")
32 CALL CHAR(95,"666666FF89FF91FF")
33 CALL CHAR(96,"2A2A2A6EFE37FFFF")
34 CALL CHAR(97,"06EFFFFFFFC3C3C3")
35 CALL CHAR(98,"3C1818383030FFFF")
36 CALL CHAR(33,"0000C0E0FEE3FE00")
37 CALL CHAR(42,"000000000000849FF")
38 CALL CHAR(104,"10383838387C7C44")
39 CALL CHAR(112,"0000001818000000")
```

8 Martian Revenge

```
40 CALL COLOR(1,7,1)
41 CALL COLOR(2,10,1)
42 CALL COLOR(8,3,1)
43 CALL COLOR(9,3,1)
44 CALL COLOR(5,16,1)
45 CALL COLOR(6,16,1)
46 CALL COLOR(7,16,1)
47 CALL COLOR(10,5,1)
48 CALL COLOR(11,8,1)
49 CALL SCREEN(2)
50 DIM A$(128)
53 FOR I=33 TO 128
54 LET A$(I)=CHR$(INT(RND*6)+93)
56 NEXT I
57 LET K=32
60 LET B$="REVENGE{4 SPACES}BOMBS{6 SPACES}HITS "
61 FOR I=1 TO LEN(B$)
62 CALL VCHAR(1,I,ASC(SEG$(B$,I,1)))
63 NEXT I
65 LET COUNT=0
66 LET SCORE=0
67 GOSUB 400
70 LET SR=2
72 LET SC=16
74 CALL VCHAR(SR,SC,33)
79 RANDOMIZE
80 FOR I=1 TO 20
82 LET A$(INT(RND*96+33))=CHR$(104)
84 NEXT I
90 LET BOMB=20
95 GOSUB 460
98 GOSUB 300
100 REM  MAIN LOOP
105 LET SPEED=SPEED+1
110 IF SPEED<INT(5-SC/8)THEN 120
115 GOSUB 300
120 CALL KEY(1,KK,S)
125 IF KK=18 THEN 500
127 CALL JOYST(1,JC,JR)
128 IF JC=4 THEN 600
129 IF JC=-4 THEN 650
130 IF .5>RND THEN 700
135 LET COUNT=COUNT+1
140 IF COUNT=5 THEN 800
190 GOTO 100
300 REM  PRINT CITY
310 FOR I=1 TO 32
320 LET A$(I)=A$(K+I)
330 CALL VCHAR(24,I,ASC(A$(I)))
```

```
340 NEXT I
350 LET K=K+1
355 LET SPEED=0
360 IF K<>96 THEN 380
370 LET K=32
380 RETURN
400 REM  SCORE
410 LET B$=STR$(SCORE)
420 FOR I=1 TO LEN(B$)
430 CALL VCHAR(1,27+I,ASC(SEG$(B$,I,1)))
440 NEXT I
450 RETURN
460 REM  BOMBS
465 LET B$=STR$(BOMB)
470 FOR I=1 TO LEN(B$)
475 CALL VCHAR(1,17+I,ASC(SEG$(B$,I,1)))
480 NEXT I
485 CALL VCHAR(1,17+I,32)
490 RETURN
500 REM  BOMBS AWAY
503 LET BOMB=BOMB-1
505 IF BOMB<0 THEN 130
506 GOSUB 460
510 FOR I=SR+1 TO 23
520 CALL VCHAR(I,SC,32)
525 CALL VCHAR(I+1,SC,112)
530 NEXT I
532 IF A$(SC)=CHR$(42)THEN 545
535 IF A$(SC)<>CHR$(104)THEN 540
537 LET SCORE=SCORE+15
540 LET SCORE=SCORE+10
545 CALL VCHAR(24,SC,42)
547 LET A$(SC)=CHR$(42)
550 LET A$(SC+K-1)=CHR$(42)
555 CALL SOUND(100,-5,0,200,0)
557 GOSUB 400
560 GOTO 130
600 REM  MOVE RIGHT
605 IF SC=32 THEN 130
610 LET SC=SC+1
615 CALL VCHAR(SR,SC-1,32)
620 CALL VCHAR(SR,SC,33)
630 GOTO 130
650 REM  MOVE LEFT
655 IF SC=1 THEN 130
660 LET SC=SC-1
665 CALL VCHAR(SR,SC+1,32)
670 CALL VCHAR(SR,SC,33)
680 GOTO 130
```

```
700 REM  MISSLES
705 IF A$(SC)<>CHR$(104)THEN 135
707 RANDOMIZE
710 LET H=INT(RND*22)
715 FOR I=22 TO H STEP -1
717 CALL VCHAR(I+1,SC,32)
720 CALL VCHAR(I,SC,104)
722 CALL SOUND(1,-6,0,800-8*I,0)
725 IF SR=I THEN 750
730 NEXT I
735 CALL VCHAR(I+1,SC,32)
740 GOTO 135
750 CALL SOUND(1000,-6,0,110,0)
752 CALL CLEAR
754 PRINT "YOU WERE HIT BY A MISSILE"
756 PRINT "YOUR SCORE WAS ";SCORE
760 GOTO 1000
800 REM  COUNT
820 IF SR=23 THEN 900
825 LET COUNT=0
830 LET SR=SR+1
835 CALL VCHAR(SR-1,SC,32)
840 CALL VCHAR(SR,SC,33)
850 GOTO 100
900 REM  CRASH
910 CALL CLEAR
920 PRINT "YOU CRASHED"
930 PRINT "YOUR SCORE WAS ";SCORE
1000 REM  END
1010 PRINT "WANT TO PLAY AGAIN"
1020 PRINT "PRESS Y FOR YES"
1030 CALL KEY(3,K,S)
1040 IF S=0 THEN 1030
1045 CALL CLEAR
1050 IF K=89 THEN 50
1060 END
```

Program Structure

Lines 25 and 26 use CALL COLOR to change all numerals to white, so they'll show on the black background when the game runs.

Lines 30-39 create the custom characters used in the game. Lines 30-35 create the buildings of the city, line 36 creates the player's ship, line 37 creates rubble that is left after a bomb has dropped, line 38 creates the missile, and line 39 draws the player's bomb figure.

Lines 40-49 color the various characters. Line 40 colors the

ship orange, while line 41 colors the rubble red. Lines 42-43 color the city characters green, lines 44-46 color the letters white for display purposes, line 47 colors the missiles purple, and line 48 colors the bomb blue. The screen color is set to black in line 49.

Lines 50-56 fill the string variable array A$ with various Martian city buildings. The array itself is used in two parts. The first 32 positions store the current line on the screen, while positions 33-128 are filled with the whole city. This creates a long cityscape that is repeated every 96 moves. As the city scrolls below the ship, new parts of the array A$ are put on the screen.

Line 57 sets the variable *K* equal to 32; this keeps track of which part of the city is on the screen. *K* is the number of the leftmost portion of the city that is currently visible.

Lines 60-63 PRINT letters on the screen. Since you can't control where PRINT places letters (it will always PRINT near the bottom), you have to use CALL VCHAR and PRINT one letter at a time by reading from a string variable, in this case B$. This technique is very useful for displaying titles and other messages as the game runs.

Line 65 sets the COUNT variable to 0, which determines when to lower the ship.

Line 66 is the variable for SCORE.

Line 67 shifts the program to the subroutine at line 400. This puts the current score on the screen.

Lines 70-72 start the ship at row 2 and column 16, while line 74 puts the ship on the screen.

Lines 79-84 put the 20 missiles in random locations into the array A$.

Line 90 sets BOMB, the counter for the number of bombs. Line 95 then GOSUBs to line 460, which displays the number.

Line 98 calls the subroutine at line 300, which prints the current part of the city visible on the screen.

Line 100 begins the main loop.

Lines 105-110 control the speed of the city as it scrolls beneath the ship. SPEED is actually a counter, and it increases to a number calculated in line 110, depending on which column the ship is in. If SC, the ship column, is large, the count is reached quickly, and the program GOSUBs to line 300, which puts the city on the screen. If SC is smaller, it takes longer to get to the maximum SPEED, so the city scrolls by slowly.

Lines 120-125 check the joystick fire button. If it was pressed, the program goes to line 500, which drops the bomb.

Lines 127-129 read the joystick. If it was pushed left, the program goes to line 650; if it was pushed right, line 600 is called.

Line 130 tests to see if a random number is less than .5. If it is, the program will go to line 700, which will fire a missile. A missile fires only half of the time.

Lines 135-140 check the COUNT variable. If it is equal to 5, the program goes to line 800, which moves the ship downward one row.

Line 190 ends the main loop.

Line 300 begins the subroutine that puts part of the city on the screen. K is a counter that points to the place in A$ where the current leftmost part of the city is.

Lines 310-340 move the city parts from the fixed part of the array (elements 33-128) into the temporary part of the array (elements 1-32). Also, the new part of the array is put on the screen in line 330.

Line 350 increases *K* so it points to the next part of the city.

Line 355 sets the SPEED counter back to 0.

Line 360 checks K. If it is less than 96, the program returns to the main loop. If it equals 96, the array resets K to 32, so it can start at the beginning of the cityscape.

The subroutine in lines 400-450 puts the score on the screen.

Lines 460-490 comprise a subroutine which displays the number of bombs left on the screen.

Line 500 begins the bomb-dropping routine. Line 503 subtracts 1 from the number of bombs left, line 505 checks to see if a bomb is left and goes to the main loop if all have been used, and line 506 GOSUBs to line 460 to print the new number of bombs left.

Lines 510-530 erase the bomb and PRINT it at the next row.

Line 532 checks to see if the bomb has hit rubble. If it has, the program goes to line 545, so that the player's score isn't increased.

Line 535 checks to see if the bomb hit a missile base. If it *didn't*, the program goes to line 540, skipping over line 537.

Line 537 adds 15 points to the score and executes only if the bomb hit a missile base.

Line 540 adds 10 points to the score. The bomb hit either a

city character or a missile, so 10 points are added for a city and a total of 25 for a missile base (15 points from line 537 + 10).

Line 545 PRINTs a rubble character on the screen, while line 547 places the rubble character into the current array position. Line 550 makes sure the rubble will reappear the next time the ship flies over it.

Line 555 creates the exploding bomb sound effect.

Line 557 GOSUBs to the subroutine which displays the new score, and line 560 returns to the main loop.

Lines 600-680 move the ship left or right, depending on joystick movement.

Line 700 begins the routine that fires the missiles from the city. Line 705 checks to see if a missile is at the same column as the ship. If it isn't, the program goes back to the main loop.

Lines 707-740 move the missile. The variable H determines how high the missile will fly before it explodes. Lines 717-722 erase the old missile, put the new one on the screen, and produce the missile's sound. Line 725 checks to see if the missile has reached the same height as the ship row. If it has, the program goes to line 750. If it hasn't, the missile continues until it reaches the height H. After that, it erases itself at its last position, and the program goes back to the main loop.

If the missile hit the ship, line 750 creates the sound of an explosion.

Lines 752-760 clear the screen and print the message. Then the program goes to line 1000 and finishes the game program.

Line 800 begins the small program that moves the ship downward one row at a time. Line 820 checks to see if the ship has reached the bottom row; if it has, the program goes to line 900. Line 825 sets COUNT back to 0. Lines 830-850 add one to the ship's row, erase the old ship, and put the new ship on the screen. Then the program goes back to the main loop.

Lines 900-930 print the crash message and your score.

Lines 1000-1060 print the final messages and reset the game. Notice that in line 1050, the program goes back to line 50, not line 10. There's no need to redefine the characters and colors again.

Variables

A$	String variable array used to store the cityscape.
K	Counter that points to the leftmost element in the cityscape array currently displayed.

B$	String which puts messages on the screen using CALL VCHAR.
COUNT	Counter to move the ship lower after a certain number of times through the loop.
SR	Ship Row.
SC	Ship Column.
BOMB	Number of bombs left.
KK	Key pressed.
S	Key status.
SPEED	Counter variable that determines how fast the city scrolls.
JC	Joystick Column direction.
JR	Joystick Row direction.

Special Notes. By using the string variable array A$, you were able to simulate a scrolling screen. The array A$ is actually split into two parts. The front end, elements 1-32, stores the current group of characters that are on the bottom of the screen. The rest of the array, elements 33-128, stores the whole city, only a part of which can be on the screen at any one time. K, a variable pointing to an element in the array, starts out at 32 and counts up each time the city "moves." When K equals 96, the K turns back to 32 and the front part fills with the earlier parts of the cityscape. This may sound confusing, but if you study it, you'll see how this array is used to simulate horizontal scrolling.

By using the SPEED variable as a special counter to determine when to scroll the city, and by checking against a variable for the ship's column, the player can vary the speed at which the city scrolls. This creates the illusion of movement.

The technique used in line 130 gives you a way to produce an inconsistent event. By using .5>RND, you are generating a number between 0 and 1; if it's less than .5, the event happens. This is often useful when you want an efficient way to include random events in your games.

Changing the Game

- Besides modifications similar to those in Chapter 7, you could change this game radically by having the ship fly past the city without crashing. You could even have more than one array and create mountain and valley characters in the bottom few rows and scrolling them together.
- You could vary the things that were in the arrays and have

different points for things you hit.

- To make the game easier or harder, you could vary the speed of scrolling, give the player more bombs, or give the Martians more missiles (or fire them more often).
- If you wanted to, you could have enemy rockets coming at the player from the right side, and you could give the player small missiles that could be fired sideways.
- You can be the attacker or the defender. By using the second joystick, you could have one person be the attacker and another be the defender.

Second City. You can make the city have a few towers sticking up into the next row so that the effect of motion will be stronger.

The towers are somewhat flimsy, and your ship will be able to ram through them, as will the enemy's missiles.

First, add these lines to the program:

```
51 DIM Z$(128)
52 GOSUB 3000
```

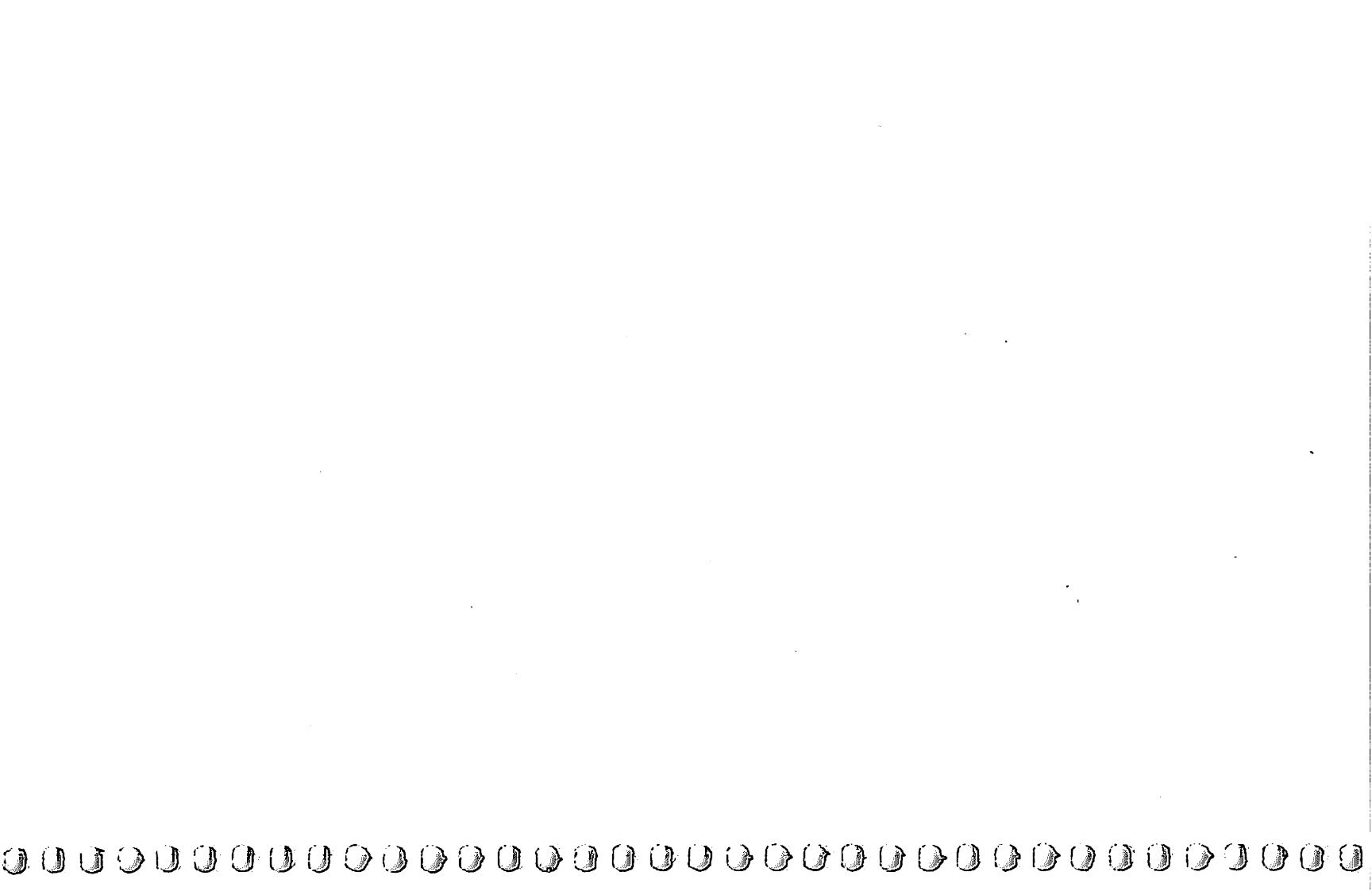
These create a new string array, Z$, which holds additional city characters in the same way that A$ did, except that Z$ will display in row 23. The second row can be created by a subroutine starting in line 3000.

```
325 LET Z$(I)=Z$(K+I)
335 CALL VCHAR(23,I,ASC(Z$(I)))
```

The above lines are used in exactly the same way as the corresponding lines for A$ (320 and 330), first to move the array elements to the screen area, and second, to put the characters on the screen in row 23.

```
3000 REM  SECOND STORY
3010 FOR I=1 TO 128
3012 LET Z$(I)=CHR$(32)
3014 NEXT I
3020 FOR I=33 TO 128
3030 IF RND>.2 THEN 3050
3040 LET Z$(I)=CHR$(98)
3050 NEXT I
3060 RETURN
```

This subroutine creates a second story for the city. The new array, Z$, is filled with spaces, then, depending on the random number from line 3030, a tower is put in the array. This should draw one tower for every five spaces, on the average.



9 Riverboat

9 Riverboat

Vertical scrolling can be used in many different types of games. In "Riverboat," it creates a river, obstacles in midstream, and even factories on the riverbank.

How to Play

Green riverbanks are on either side, and your riverboat appears at the top of the screen as the game begins.

You always move downriver, the vertical scrolling making it seem as if the boat moves, while it is actually the scenery which changes position beside your boat.

You may move the boat to the left or right with the joystick, but be careful not to crash into the riverbanks, which are always changing as the river winds back and forth.

Once in a while you'll see a blue factory with smokestacks and a dock attached to it. Don't crash into the factory or the docks!

If you move alongside a dock, press the joystick fire button and pick up cargo from the docks. If you do it correctly, your score will show on the left side of the screen. If you miss the cargo, the word *oops* appears instead.

The river goes on forever. If you crash into the bank or a dock, a message appears, telling you how many times you successfully loaded cargo.

Program 9-1. Riverboat

```
10 REM RIVERBOAT
20 CALL CLEAR
25 CALL SCREEN(8)
30 CALL COLOR(1,7,1)
32 CALL COLOR(2,13,1)
36 CALL COLOR(9,5,1)
40 CALL CHAR(33,"FFFFFF7E7E7E3C18")
42 CALL CHAR(34,"7E7E7E00FFFFFFFF")
44 CALL CHAR(42,"FFFFFFFFFFFFFFFF")
46 CALL CHAR(96,"2A2A2A6EFEE7FFFF")
48 CALL CHAR(97,"00000000000003FF")
50 LET RR=3
60 LET LB=11
62 LET RB=21
64 FOR I=1 TO 3
65 GOSUB 300
66 NEXT I
69 LET RC=INT((RB+LB)/2)
```

9 Riverboat

```
70 FOR I=1 TO 20
75 GOSUB 300
78 NEXT I
80 GOSUB 200
90 LET SCORE=0
100 REM MAIN LOOP
110 GOSUB 400
115 GOSUB 700
120 GOSUB 200
125 GOSUB 900
130 GOSUB 300
190 GOTO 100
200 REM PRINT BOAT
210 CALL VCHAR(RR-2,RC,32)
220 CALL VCHAR(RR-1,RC,34)
230 CALL VCHAR(RR,RC,33)
232 CALL SOUND(100,-5,0,910,15)
233 CALL SOUND(200,-5,15,910,20)
235 RETURN
300 REM PRINT BANK
305 RANDOMIZE
310 CALL HCHAR(24,1,42,LB)
320 CALL HCHAR(24,RB,42,32-RB)
322 IF RND<.9 THEN 330
324 IF RND<.5 THEN 328
325 CALL VCHAR(24,LB+1,96)
326 CALL VCHAR(24,LB+2,97)
327 GOTO 330
328 CALL VCHAR(24,RB-1,96)
329 CALL VCHAR(24,RB-2,97)
330 LET RB=RB+INT(RND*3)-1
340 LET LB=LB+INT(RND*3)-1
342 IF RB-LB<6 THEN 380
343 IF LB<7 THEN 360
344 IF RB>28 THEN 370
345 PRINT
350 RETURN
360 LET LB=LB+1
365 GOTO 345
370 LET RB=RB-1
375 GOTO 345
380 LET LB=LB-1
382 LET RB=RB+1
385 GOTO 345
400 REM PICK UP CARGO
410 CALL KEY(1,K,S)
420 IF K<>18 THEN 490
430 CALL GCHAR(RR-2,RC-1,C)
431 IF C=97 THEN 440
```

```
432 CALL GCHAR(RR-1,RC-1,C)
433 IF C=97 THEN 440
434 CALL GCHAR(RR-1,RC+1,C)
435 IF C=97 THEN 440
436 CALL GCHAR(RR-2,RC+1,C)
437 IF C<>97 THEN 500
440 LET SCORE=SCORE+1
445 CALL SOUND(200,440,0)
450 LET A$=STR$(SCORE)
455 LET COL=1
460 GOSUB 600
490 RETURN
500 REM PRINT ERROR
510 LET A$="OOPS"
520 LET COL=1
530 GOSUB 600
535 CALL SOUND(1000,-1,0,200,0)
560 RETURN
600 REM PRINT MESSAGE
610 FOR I=1 TO LEN(A$)
620 CALL VCHAR(23,COL+I,ASC(SEG$(A$,I,1)))
630 NEXT I
640 RETURN
700 REM BOAT JOYST
710 CALL JOYST(1,JC,JR)
720 IF JC=4 THEN 750
722 IF JC=-4 THEN 780
730 RETURN
750 CALL GCHAR(RR,RC+1,C)
752 CALL GCHAR(RR-1,RC+1,C)
754 IF C<>32 THEN 800
760 LET RC=RC+1
762 CALL VCHAR(RR-2,RC-1,32,2)
765 RETURN
780 CALL GCHAR(RR,RC-1,C)
782 CALL GCHAR(RR-1,RC-1,C)
784 IF C<>32 THEN 800
790 LET RC=RC-1
792 CALL VCHAR(RR-2,RC+1,32,2)
795 RETURN
800 REM CRASH
805 FOR I=1 TO 10
806 CALL SOUND(300,-6,I-1)
807 NEXT I
810 PRINT "YOUR BOAT CRASHED"
820 PRINT "YOUR SCORE WAS ";SCORE
830 PRINT "WANT TO PLAY AGAIN"
840 PRINT "PRESS Y FOR YES"
850 CALL KEY(3,K,S)
```

```
860 IF S=0 THEN 850
870 IF K=89 THEN 10
880 END
900 REM CHECK AHEAD
910 CALL GCHAR(RR+1,RC,C)
920 IF C<>32 THEN 800
930 RETURN
```

Program Structure

Line 20 clears the screen, and line 25 colors the screen blue.

Line 30 colors the riverboat dark red, line 32 colors the riverbanks dark green, and line 36 colors the factories and docks dark blue.

Lines 40-48 create the custom characters. Lines 40-42 create the riverboat, line 44 the riverbank, and lines 46-48 the factory and dock.

Line 50 makes the riverboat's row position equal to 3.

Lines 60-62 make the initial left bank column equal to 11 and the initial right bank column equal to 21.

Lines 64-66 set up a FOR/NEXT loop to run three times to print the first three segments of the riverbank; this is needed so that the riverboat doesn't crash into the riverbank right away.

Line 69 sets the riverboat column to the middle of the first three segments of the riverbank.

Lines 70-78 create the rest of the riverbank, adding 20 more segments.

Line 80 uses a GOSUB to put the riverboat in the water, and line 90 sets the SCORE to 0.

Lines 100-190 are the main loop. You'll notice that in this game, the main loop uses nothing but GOSUB to subroutines. This creates a cleaner game program, and makes the order of the main loop elements easier to change. By looking to the called subroutines, you can see what each main loop line does.

Lines 200-235 comprise the subroutine which puts the boat on the screen and create the sound effect of the riverboat's paddlewheels. The boat is actually composed of three characters: the bow, the midship, and the stern, which is actually a space. By putting a space at the end, you can easily make the boat *seem* to move.

In lines 300-385 is a subroutine which puts the riverbank's next segment at the bottom of the screen and then scrolls it upward with a PRINT statement. This subroutine is the heart

of the vertical scrolling technique.

Line 305 uses RANDOMIZE to make sure that each bank is randomly drawn.

Lines 310-320 put the left and right banks on the screen. The value was calculated the last time this subroutine was called. By using HCHAR, and having several characters PRINTed at once, you can create the left and right bank quickly. The left bank starts at column 1 and goes to the column number stored in the variable LB. The right bank starts out at the column number in variable RB and PRINTs characters to the right edge of the screen.

Line 322 generates a random number and checks to see if it is less than .9. If it is, the program goes to line 330. If it isn't, the program continues to line 324, which puts a factory on the riverbank an average of one out of every ten times.

Line 324 generates another random number to determine whether a factory will be put on the right or the left bank. If it is less than .5, the program goes to line 328. If it isn't, the program continues.

Lines 325-327 put the factory and the dock on the left bank and send the program to line 330, while lines 328-329 do the same on the right bank.

Lines 330-340 create new values for the right and left banks by adding 1 to or subtracting 1 from each bank's value.

Lines 342-344 check the left and right bank (LB, RB) values for various operations.

Lines 360-365 take care of what happens if the left bank was less than 7. One is added to the left bank, and the program goes to line 345 to complete the riverbank subroutine. Lines 370-375 take care of what happens if the right bank is greater than 28. One is subtracted from the right bank and the program goes to line 345.

If the distance between the two banks is too small, lines 380-385 are executed. One is added to the left bank and one is subtracted from the right bank.

Lines 400-490 execute when the joystick fire button is pressed. Lines 410-420 check the fire button. Lines 430-437 use CALL GCHAR to check for a dock next to either side of the riverboat's midship or bow. Lines 430, 432, 434, and 436 test each of the four possibilities. Lines 431, 433, and 435 test to see if the dock character is there. If it is, the program goes to line 440. If if gets all the way through without finding any docks,

line 437 makes the program go to line 500, which tells you that you missed.

Line 440 adds one point to your score, and line 445 makes a sound to tell you that you were successful.

Lines 450-460 set up the call to the subroutine in line 600, which displays your score.

Line 490 RETURNs the program to the main loop.

Lines 500-560 continue part of the subroutine that started in line 400. A$ and COL are given the proper values to print *OOPS* on the screen, and the subroutine at line 600 puts the message on the screen.

The subroutine in lines 600-640 can be called from anywhere to put a message on the screen. This subroutine needs a column, which it finds in the variable COL, and a string variable, A$, which contains the message or score you want put on the screen. By setting up a message subroutine, you can print scores, messages, or anything you want at any place on the screen.

Lines 700-795 are a subroutine that moves the boat right or left, depending on whether you have pushed the joystick or not. This is similar to other joystick routines you've seen in this book. The riverboat's column is reduced or increased by one, depending on the joystick direction, and its old location is erased.

Lines 750-754 check for a crash when the riverboat moves right, and lines 780-784 check for a crash when moving left.

Lines 800-880 are called if the boat does crash.

Lines 805-807 use FOR/NEXT loop to create a sound that plays ten times, but softer each time, to simulate a large explosion.

Lines 810-880 should be familiar. They PRINT a message, the score, and provide an option to play again.

The subroutine in lines 900-930 checks for a head-on collision. Lines 910-920 use CALL GCHAR to test the row and column immediately in front of the bow. If it is not water (ASCII 32), there's a crash and the program goes to line 800. Otherwise, the program RETURNs to the main loop.

Variables

RR	Riverboat Row.
RC	Riverboat Column.
LB	Left Bank column.

RB	Right Bank column.
K	Key pressed.
S	Key Status.
C	Variable used to store the character value found by CALL GCHAR.

Special Notes. This program uses the PRINT statement to create a scrolling river, simulating vertical movement.

The subroutine at line 600 is an example of a generalized message display routine. You must put a column number into COL and a message into the string variable A$ *before* you call this subroutine. Whatever you put into these will be displayed on the screen in the column you selected.

This program takes several safety measures. It checks collisions in front and on the sides, and makes sure that the riverbanks are within certain values so that the river doesn't run off the screen or get too narrow. The program must also check all possibilities for getting cargo, and put a message on the screen if you miss. Try to think of every possibility and have the computer check for it. By setting limits in the creation of the riverbanks, for example, you make sure that the game is playable.

Putting a space behind a character, as is done with the riverboat, insures that when the screen scrolls, and the new character appears, it doesn't need a separate erasure.

Extended BASIC. If you have the TI Extended BASIC cartridge, you'll have to modify this program slightly. In Extended BASIC, the PRINT statement starts printing one row higher than in TI BASIC. Instead of printing at row 24, it prints at 23.

To change the program to make it work in Extended BASIC, simply change the row values that everything starts at. Change lines 310, 320, 325, 326, 328, and 329 so that the row number is 23 instead of 24. Change line 620 so that the row number is 22 instead of 23. Finally, change line 70 so that the FOR/NEXT loop counts only to 19 instead of 20.

Changing the Game

- You can make the game more interesting by allowing the riverboat to move forward or backward, and adjusting how often the river scrolls forward by the method you saw for changing scrolling speed in "Martian Revenge."
- You could also add a different boat at the bottom, one able

to move upward. Allow each boat to fire, and have the second boat controlled by joystick 2. You could then have a riverboat battle between two players.

- Of course, you can have different factories with different score values, or you could add floating mines that the riverboat would have to avoid.
- You could put guns on the riverbanks and have them fire at the riverboat and the riverboat fire back. You could create drawbridges that would need to be raised for the riverboat to pass; the only way to do so might be to dock a few rows before the boat reaches the bridge.
- If you allow your riverboat to back up, you could also create islands which split the river in two, and have one of the side rivers dead-end so that the riverboat has to back up before it crashes.
- Finally, you could change the game by having the water in the river of varying depths, making it more complicated. Some depths could be safe, others so low that the boat might (depending on a random number) get stuck.
- You could dress up this world by putting houses and forests on each side, or other boats in the river for you to avoid.

Sandbars. You can change the game so that the river has sandbars to make it more difficult to travel. At the same time, by adding a few more lines, you can keep track of how far the riverboat has traveled.

To start, you could add these lines:

```
55 CALL CHAR(116,"AA55AA55AA55AA55")
56 CALL COLOR(11,11,1)
57 LET DIST=Ø
```

These lines create the shape and color of the sandbar, and set the initial DIST variable to 0, which is used to calculate the distance.

```
3Ø4 LET DIST=DIST+1
```

The above line increases the distance every time a riverbank is put on the screen.

```
321 IF RND<.1 THEN 3ØØØ
```

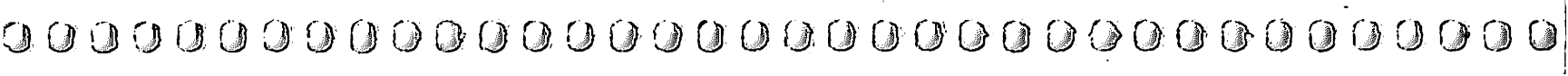
If the random number is less than .1, the program goes to line 3000, which puts a sandbar in a river.

```
825 PRINT "YOU WENT ";(DIST-2Ø)/2Ø;"MILES"
```

displays the final message so that you can see how far the boat got.

```
3000 REM  PUT SANDBAR IN RIVER
3005 IF DIST<10 THEN 322
3010 LET BAR=(LB+RB)/2
3020 CALL VCHAR(24,BAR,116)
3030 GOTO 322
```

This routine puts a sandbar in the river. If the DIST variable is less than 10, no sandbar is PRINTed; this might make the start of the game too difficult. Line 3010 calculates the center of the river, and line 3020 puts the sandbar in the river.



10 Shark

10 Shark

If you follow the river of no return long enough, you'll eventually reach the ocean, and there you'll find sharks.

A target practice game that shows how to create complex automatic motion, "Shark" also shows you a different way to use joysticks to shoot at a target.

How to Play

The game starts out with the creation of a playing field. Your mission is to stun a shark with your electro-zap gun. The shark spends most of its time underwater, but due to the quirks of the electro-zap gun, you can stun it more easily if you hit it while it's underwater.

If you stun the shark while it's underwater, you get 25 points; if it's on the surface, you'll get only 5 points. You need at least 200 points to immobilize the shark.

The shark starts out at the upper-left corner of the screen, then swims to the right and left side, making a slow zigzag pattern as it wends its way to the bottom of the screen.

If the shark reaches the bottom, it starts again at the top-left corner.

Your score is displayed along the right border of the screen as a striped bar chart. When you hit the shark on its fin, you hear one kind of sound. When you hit it underwater, you'll hear another. If you miss, you'll see your shot hit the water and hear yet another sound.

You aim by setting the row and column pointers, which are along the bottom and right sides of the screen, to match the row and column of the shark. When you're ready, press the fire button. After you fire, there'll be a short delay while the electro-zap gun recharges.

Program 10-1. Shark

```
10 REM  SHARK
20 CALL CLEAR
25 CALL SCREEN(6)
28 CALL CHAR(42,"FFFFFFFFFFFFFFFF")
29 CALL CHAR(104,"3F003F003F003F00")
30 CALL CHAR(33,"80C0E0F0F8FCFEFF")
31 CALL CHAR(97,"0101011FFFF0101011")
32 CALL CHAR(34,"0103070F1F3F7FFF")
33 CALL CHAR(96,"181818181818FFFF")
```

10 Shark

```
34 CALL COLOR(10,16,1)
35 CALL COLOR(2,16,1)
39 CALL COLOR(9,7,16)
40 CALL VCHAR(1,30,42,24)
41 CALL HCHAR(24,1,42,30)
60 LET CC=15
62 LET CR=12
65 CALL VCHAR(24,CC,96)
66 CALL VCHAR(CR,30,97)
70 LET FLAG=1
80 LET SR=1
82 LET SC=2
87 LET SCORE=0
88 LET COUNT=0
90 LET SLOW=0
95 LET SHOTS=0
97 LET DELAY=0
100 REM  MAIN LOOP
105 LET DELAY=DELAY+1
110 GOSUB 300
115 LET SLOW=SLOW+1
117 IF SLOW=3 THEN 119
118 GOTO 130
119 LET SLOW=0
120 IF FLAG=1 THEN 700
125 IF FLAG=0 THEN 750
130 CALL KEY(1,K,S)
135 IF K=18 THEN 800
190 GOTO 100
200 REM TIME
210 FOR I=1 TO 250
220 NEXT I
230 RETURN
300 REM JOYSTICK
310 CALL JOYST(1,JC,JR)
315 IF JR=4 THEN 400
320 IF JR=-4 THEN 450
325 IF JC=4 THEN 500
330 IF JC=-4 THEN 550
335 RETURN
400 REM  MOVE UP
405 IF JC<>0 THEN 335
410 IF CR=1 THEN 335
420 LET CR=CR-1
430 CALL VCHAR(CR+1,30,42)
435 CALL VCHAR(CR,30,97)
440 RETURN
450 REM  MOVE DOWN
455 IF JC<>0 THEN 335
```

```
460 IF CR=23 THEN 335
470 LET CR=CR+1
480 CALL VCHAR(CR-1,30,42)
485 CALL VCHAR(CR,30,97)
490 RETURN
500 REM  MOVE RIGHT
505 IF JR<>0 THEN 335
510 IF CC=29 THEN 335
520 LET CC=CC+1
530 CALL VCHAR(24,CC-1,42)
535 CALL VCHAR(24,CC,96)
540 RETURN
550 REM  MOVE LEFT
555 IF JR<>0 THEN 335
560 IF CC=2 THEN 335
570 LET CC=CC-1
580 CALL VCHAR(24,CC+1,42)
585 CALL VCHAR(24,CC,96)
590 RETURN
700 REM MOVE SHARK RIGHT
702 LET COUNT=COUNT+1
705 IF SC=28 THEN 740
710 LET SC=SC+1
715 RANDOMIZE
720 IF RND>.2 THEN 737
730 CALL VCHAR(SR,SC,33)
737 CALL VCHAR(SR,SC-1,32)
738 GOTO 130
740 LET SR=SR+1
742 CALL VCHAR(SR-1,SC,32)
744 LET FLAG=0
745 IF SR=24 THEN 850
747 GOTO 130
750 REM MOVE SHARK LEFT
752 LET COUNT=COUNT+1
755 IF SC=2 THEN 790
760 LET SC=SC-1
765 RANDOMIZE
770 IF RND>.2 THEN 787
780 CALL VCHAR(SR,SC,34)
787 CALL VCHAR(SR,SC+1,32)
788 GOTO 130
790 LET SR=SR+1
792 CALL VCHAR(SR-1,SC,32)
794 LET FLAG=1
797 GOTO 130
800 REM  FIRE AT SHARK
801 IF DELAY<10 THEN 100
802 LET SHOTS=SHOTS+1
```

10 Shark

```
803 LET DELAY=0
804 CALL GCHAR(CR,CC,C)
805 IF C<>32 THEN 810
806 CALL VCHAR(CR,CC,46)
807 GOSUB 200
808 CALL VCHAR(CR,CC,32)
809 CALL SOUND(10,-6,0)
810 IF SR<>CR THEN 100
815 IF SC<>CC THEN 100
825 IF C<>32 THEN 830
826 CALL SOUND(200,110,0)
827 LET SCORE=SCORE+20
830 LET SCORE=SCORE+5
832 CALL SOUND(100,880,0)
835 IF SCORE>200 THEN 1000
836 GOSUB 900
837 GOTO 100
850 REM  SHARK STARTS OVER
851 LET A$="SHARK STARTS OVER"
855 FOR I=1 TO LEN(A$)
860 CALL VCHAR(1,3+I,ASC(SEG$(A$,I,1)))
862 CALL SOUND(100,-2,0,440,0)
865 NEXT I
870 GOSUB 200
880 CALL HCHAR(1,3,32,3+LEN(A$))
885 LET SR=1
886 LET SC=2
887 LET FLAG=1
890 GOTO 130
900 REM PRINT SCORE
910 CALL VCHAR(1,31,104,INT(SCORE/10))
920 RETURN
1000 REM  GAME OVER
1010 CALL CLEAR
1015 CALL COLOR(9,2,1)
1020 PRINT "YOU GOT THE SHARK"
1030 PRINT "IT TOOK YOU ";SHOTS;" SHOTS"
1040 PRINT "AND THE SHARK SWAM ";COUNT;" YARDS"
1050 PRINT "WANT TO PLAY AGAIN"
1060 PRINT "PRESS Y FOR YES"
1070 CALL KEY(3,K,S)
1080 IF S=0 THEN 1070
1090 IF K=89 THEN 10
1095 END
```

Program Structure

Line 20 clears the screen, and line 25 colors the screen light blue.

Lines 28-33 create the character shapes: the slot the pointers fit in for the electro-zap gun is drawn in line 28; the right-facing and left-facing shark fins in lines 30 and 32; the column and row pointers in lines 31 and 33; while line 29 creates the bar chart.

Lines 34-39 color the bar chart white, the pointer slots white, and the pointers dark red.

Using CALL VCHAR and CALL HCHAR, lines 40-41 create the slots that the pointers fit in.

Lines 60-62 create the beginning values of the pointers, with the row and column of 12 and 15. Lines 65-66 then put the pointers on the screen.

Lines 70-97 initialize the variables used in the game. Refer to the variable list at the end of this section.

Lines 100-190 are the main loop. Line 105 increases the DELAY variable by 1. This increases each time through the loop, and is used to see if enough time has passed to allow the gun to fire. If the joystick fire button is pushed, DELAY is checked to see if it is 10 or more. After it's fired, DELAY is set back to 0 again, and the count resumes.

Lines 115-125 move the shark every three times through the loop and move it to the right or left depending on how the FLAG variable is set. The computer-controlled movement of the shark is a bit different than that used previously.

Line 115 adds 1 to the variable SLOW and line 117 sees if SLOW is equal to 3. If it is, the program goes to line 119. If it isn't, the program skips to line 130. Line 119 resets SLOW to 0, and lines 120 and 125 check to see whether the FLAG variable is 1 or 0, moving the shark right or left through later subroutines.

Lines 200-230 create a time delay using a FOR/NEXT loop. Delay loops can be placed in subroutines so they can be called whenever you need to slow the program down.

Our standard joystick subroutine is in lines 300-335.

Lines 400-590 move the row and column electro-zap gun pointers up, down, right, or left.

Line 410 checks the pointer row (CR). If it's equal to 1, the pointer cannot go any higher. Line 420 subtracts 1 from the row pointer.

Lines 430-435 erase the pointer at its old location and put it on the screen at its new position. Notice, in erasing, that the slot (ASCII 42) is put on the screen, not a space, and that the

pointer has a white background to match the color of the slot.

Similarly, lines 450-590 perform the functions for down, right, and left moves of the pointers.

Lines 700-747 move the shark right. This was gone to from line 120 if the FLAG was 1. Line 702 adds 1 to the COUNT variable and line 705 checks to see if the shark column is equal to 28. If it is, the shark cannot swim any further to the right. Line 710 then adds 1 to the shark column.

If the random number generated in line 720 is greater than .2, the program goes to line 737, which makes sure the shark's fin is *not* printed. If the random number is less than .2, the fin is PRINTed and erased.

Line 740 starts the sequence used to turn the shark around. First the shark's row is increased by 1, then line 742 erases the old character if it happened to be on the screen. Line 744 sets the FLAG to 0, which tells the main loop that the shark is now swimming from right to left. Finally, line 745 checks to see if the shark row is 24, which is as far as it can go down. If it is, the program goes to line 850, which starts the shark over again.

Lines 750-797 move the shark left.

Lines 800-837 check to see what happens when the gun is fired.

If the DELAY variable is less than 10, the program goes back to the main loop. Line 802 adds 1 to the SHOTS variable, and line 803 resets the DELAY variable to 0, so the main loop executes 10 times before the gun is fired again.

Line 804 sees if anything is in the position pointed to by the row and column gun pointers. Line 805 sees if it is not an empty space. If it's not a space (checked in line 805), the program goes to line 810, indicating that you stunned the shark.

Line 806 lets you see where your shot landed. Line 808 erases the shot, and line 809 makes the sound of the shot hitting the water.

Lines 810-815 test to see if the shark's row and column match the pointers' row and column.

Line 825 again uses the variable C; if C is not equal to 32 (space), the shark was hit and the program goes to line 830.

If a fin was not hit, but you have hit the shark, line 826 prints the victory sound for hitting the submerged shark, and line 827 adds 20 to your score. The program goes to line 830,

where 5 points are added to the score if you hit the shark on the surface.

Line 835 checks to see if the SCORE variable is greater than 200. If it is, the shark has been immobilized and the program goes to line 1000.

Lines 850-890 reset the shark if it has gone all the way to the bottom-right corner.

A message shows on the screen so you'll know that the shark is back at its starting place. Also, a warbling tone is called from line 862.

Line 885-887 reset the shark row and column to the beginning values and the FLAG variable to 1.

Lines 900-920 comprise a subroutine that puts the score on the screen as a bar chart created with CALL VCHAR.

The ending routine is in lines 1000-1095. It PRINTs a message telling you that you immobilized the shark, how many shots it took, and how far the shark swam. Then it offers another chance to play the game.

Variables

CC	Column pointer for the gun.
CR	Row pointer for the gun.
FLAG	Variable used to tell whether the shark is swimming left or right. If it is swimming right, FLAG = 1; if it is swimming left, FLAG = 0.
SR	Shark's Row.
SC	Shark's Column.
COUNT	Distance that the shark travels.
SLOW	Counter variable used to move the shark three times through the main loop.
SHOTS	Stores the number of shots fired.
DELAY	Makes sure the gun won't fire until the main loop has executed at least 10 cycles.
K	Key pressed.
S	Key Status.
JR	Joystick Row.
JC	Joystick Column.
C	Character value that CALL GCHAR finds in a specific row and column.

Special Notes. The row and column pointers demonstrate a different way to choose a row and column by pushing the joysticks.

10 Shark

The other main feature of this game is the computer-controlled motion of the shark. It moves in a known pattern, but because most of its trip is unseen, the game is different from a normal shooting-style game. The shark moves until it gets to a certain point, then it moves in a different direction. By using this kind of patterned motion, you can create all kinds of complicated characters which can take on personalities of their own. Notice the use of FLAG to tell the computer which way something is going. By using more complex flags, you can control characters in any way you want.

Notice the sequence in lines 826-832. By adding scores and sounds together, you can combine elements to create a total that is either a single part or a total of both parts. If one thing happens, one score and sound are obtained, but if another event occurs, both scores and both sounds display and execute.

Changing the Game

- You could fill the ocean with different kinds of creatures, each with different points awarded for stunning them.
- This game could be modified so that the shark would be on the surface all the time, but moving at varying speeds.
- You could, instead, have another character that appears on the screen, for instance, a dolphin. If the shark appears, you could stun it for points, but a dolphin, which would have a fin of a different color, would have to left alone, or you would lose points. You'd have to decide quickly.
- You could also change the game so that you'd have to manipulate a third dimension. By pressing the joystick fire button *and* moving the joystick up or down, you could adjust the depth that your shot fired. Making the shark go to different depths and awarding points for close shots could also be included.
- You could turn this into a fast-action, two-player game by having a second joystick with the pointers on the top and left sides of the screen, and see who could get the shark first.
- Add more sound to this game, by producing one kind of splash when the shark surfaces and another when it dives back below the water.

Dolphin. You can add a few lines to make it so a dolphin is swimming with the shark. Sometimes the shark will be on

the surface, other times the dolphin. If you hit a dolphin, you'll lose your score.

To add the dolphin, begin adding these lines to Shark:

```
24 CALL COLOR(11,11,1)
26 CALL CHAR(112,"80C0E0F0F8FCFEFF")
27 CALL CHAR(113,"0103070F1F3F7FFF")
```

The dolphin's fin is the same shape as the shark's, but is yellow.

```
102 GOSUB 3000
```

Adding the above line to the main loop sends the program to the subroutine at line 3000.

```
817 IF DF=0 THEN 825
819 CALL VCHAR(1,31,32,24)
820 LET SCORE=0
822 CALL VCHAR(SR,SC,32)
823 LET A$="YOU HIT A DOLPHIN"
824 GOTO 855
```

Insert these lines to check if you hit a dolphin. If DF is 0, the shark is on the surface. If not, lines 819-824 take care of what happens if you hit a dolphin.

Line 819 erases your score, line 820 sets the SCORE variable to 0, line 822 erases the dolphin on the screen, and line 823 sets up the message to tell you that you hit a dolphin.

Line 824 sends the program to line 855, where the message is printed and the game starts over again.

```
3000 REM  SWITCH
3010 IF DF=1 THEN 3020
3012 LET DO=112
3013 LET DI=113
3015 LET DF=1
3017 GOTO 3030
3020 LET DF=0
3022 LET DO=33
3024 LET DI=34
3030 RETURN
```

To change the shark and dolphin positions, you'll need the lines above. If DF is 1, the dolphin is on the surface, and the program goes to line 3020.

If DF is not 1, DO and DI are set to give the dolphin shapes, and DF is set to 1 so that the dolphin will be on the surface.

If DF = 1, the shark's shape, DO and DI are set, and DF is set to 0 so that the shark can move on the surface.

You'll also need to change these lines:

```
730 CALL VCHAR(SR,SC,DO)
780 CALL VCHAR(SR,SC,DI)
```

These lines put the fin above the water; if DO is 32 and DI is 33, a shark is printed; if DO is 112 and DI is 113, a dolphin is shown.

11 Mushrooms

11 Mushrooms

The caterpillar we designed earlier is back—now it's demonstrating a technique which makes it seem to grow and grow and grow. You'll see how to create a completely different kind of motion, one for a creature that's always moving.

How to Play

A red-fenced garden for the caterpillar to crawl around in is drawn, and the title and score appear. The caterpillar soon starts crawling around the garden.

Unlike the other objects you've manipulated, the caterpillar is always moving. You affect the *direction* of the motion by moving the joystick. If you push it left, the caterpillar changes direction to move left; a right joystick move makes the caterpillar go right, and so on.

Be careful. If you are moving left and pull the stick to the right, the caterpillar will crash into itself.

This is a hungry caterpillar, and it eats mushrooms. As the game continues, more and more mushrooms sprout in the garden. Beware! Only the red mushrooms are good to eat. Blue mushrooms are poison to your caterpillar.

Steer the caterpillar carefully. If it hits the red wall at the edge of the garden, it will crash, and if it hits any of its own body, it will also crash.

One more thing: Each time the caterpillar eats a mushroom, it grows longer. How long can your caterpillar get before it runs out of room?

When you crash, your score is shown, and you can play again.

Program 11-1. Mushrooms

```
10 REM MUSHROOMS
20 CALL CLEAR
25 CALL SCREEN(15)
30 CALL CHAR(33,"143E2A7F436B3E36")
32 CALL CHAR(34,"00EEBFFDEFFFAAAA")
34 CALL CHAR(42,"CCCCFFFFFFFFFFFF")
35 CALL CHAR(43,"FFFFFFFFFFFFFFFF")
37 CALL CHAR(104,"3CFFFF9918181818")
38 CALL CHAR(112,"3CFFFF9918181818")
40 CALL COLOR(1,13,1)
```

11 Mushrooms

```
44 CALL COLOR(2,9,1)
47 CALL COLOR(10,14,1)
48 CALL COLOR(11,5,1)
52 CALL HCHAR(2,1,42,32)
53 CALL HCHAR(23,1,42,32)
57 CALL VCHAR(3,1,43,20)
58 CALL VCHAR(3,32,43,20)
60 LET A$="MUSHROOMS{8 SPACES}p = POISON"
61 LET ROW=1
62 LET COL=2
64 GOSUB 200
65 LET SCORE=0
66 LET A$="SCORE{3 SPACES}"&STR$(SCORE)
67 LET ROW=24
68 LET COL=10
69 GOSUB 200
70 LET COL=18
80 LET COUNT=0
82 DIM R(500)
83 DIM C(500)
85 LET CR=12
86 LET CC=16
87 LET R(1)=CR
88 LET C(1)=CC
90 LET D=1
97 LET K=1
98 LET M=0
100 REM MAIN LOOP
110 LET COUNT=COUNT+1
112 IF COUNT=3 THEN 300
120 CALL JOYST(1,JC,JR)
122 IF JR=4 THEN 500
123 IF JC=4 THEN 510
124 IF JR=-4 THEN 520
125 IF JC=-4 THEN 530
130 IF M>0 THEN 800
140 LET M=M+1
190 GOTO 100
200 REM MESSAGE
210 FOR I=1 TO LEN(A$)
220 CALL VCHAR(ROW,COL+I,ASC(SEG$(A$,I,1)))
230 NEXT I
240 RETURN
300 REM UPDATE MOVE
305 LET COUNT=0
307 CALL VCHAR(CR,CC,34)
310 ON D GOTO 320,330,340,350
320 LET CR=CR-1
325 GOTO 360
```

```
330 LET CC=CC+1
335 GOTO 360
340 LET CR=CR+1
345 GOTO 360
350 LET CC=CC-1
360 CALL GCHAR(CR,CC,N)
365 IF N=32 THEN 380
367 IF N=112 THEN 900
370 IF N<>104 THEN 400
372 LET SCORE=SCORE+1
373 LET A$=STR$(SCORE)
374 GOSUB 200
380 CALL VCHAR(CR,CC,33)
381 LET K=K+1
382 LET R(K)=CR
384 LET C(K)=CC
387 IF K>SCORE+3 THEN 390
388 GOTO 300
390 CALL VCHAR(R(1),C(1),32)
391 LET K=K-1
392 FOR I=1 TO K
393 LET R(I)=R(I+1)
394 LET C(I)=C(I+1)
395 NEXT I
397 GOTO 120
400 REM CRASH
405 CALL CLEAR
410 PRINT "YOU CRASHED"
420 GOTO 930
500 REM MOVE UP
501 IF JC<>0 THEN 130
502 LET D=1
505 GOTO 130
510 REM MOVE RIGHT
512 IF JR<>0 THEN 130
513 LET D=2
515 GOTO 130
520 REM MOVE DOWN
522 IF JC<>0 THEN 130
523 LET D=3
525 GOTO 130
530 REM MOVE LEFT
532 IF JR<>0 THEN 130
533 LET D=4
535 GOTO 130
800 REM PUT DOWN MUSHROOMS
810 LET M=0
815 RANDOMIZE
820 LET MR=INT(RND*21)+2
```

11 Mushrooms

```
825 LET MC=INT(RND*30)+2
830 CALL GCHAR(MR,MC,N)
840 IF N<>32 THEN 820
842 IF RND<.7 THEN 845
843 CALL VCHAR(MR,MC,112)
844 GOTO 100
845 CALL VCHAR(MR,MC,104)
850 GOTO 100
900 REM POISON
910 CALL CLEAR
920 PRINT "YOU ATE A POISON MUSHROOM "
930 PRINT "YOUR SCORE WAS ";SCORE;" MUSHROOM
    S"
940 PRINT "WANT TO PLAY AGAIN"
950 PRINT "PRESS Y FOR YES"
960 CALL KEY(3,K,S)
970 IF S=0 THEN 960
980 IF K=89 THEN 10
990 END
```

Program Structure

Line 20 clears the screen, and line 25 colors it gray.

Lines 30-38 create the custom characters. Line 30 creates the head of the caterpillar, line 32 the body, line 34 the top and bottom of the garden fence, line 35 the sides, line 37 the edible mushroom, and line 38 the poison mushroom.

Line 40 colors the caterpillar green and line 44 colors the fence red. Line 47 colors the edible mushrooms dark red, while line 48 colors the poison mushrooms blue.

Lines 52-58 use HCHAR and VCHAR to create the garden fence.

Lines 60-64 put the title and warning about poison mushrooms on the screen. A generalized message subroutine that starts at line 200 is used here. You supply the message in the string variable A$, a value for the variable ROW, and a value for variable COL.

Lines 66-69 put the current score on the screen. Notice how the ampersand (&) is used to add two strings together.

Line 70 sets COL to 18 to PRINT the score.

Line 80 sets COUNT to 0. COUNT will check to see how often the caterpillar moves.

Lines 82-83 create variable arrays R and C, which store the current rows and columns of *all* the caterpillar's segments.

Lines 85-86 create the row and column variables that will hold the row and column values of the caterpillar's head, and

lines 87-88 put the caterpillar's head row and column into the first elements of the arrays.

Line 90 sets D equal to 1. D is the direction that the caterpillar moves. If D is equal to 1, the caterpillar goes up; 2 moves it right, 3 for down, and 4 for left.

Line 97 sets K equal to 1. K acts as a pointer to show how long the caterpillar is and where its head is in the array that stores the body parts.

Line 98 sets M equal to 0. M is a counter used to PRINT mushrooms on the screen.

Lines 110-112 increase COUNT by 1. If COUNT is equal to 3, the program goes to line 300, which moves the caterpillar.

Lines 120-125 read the joystick. Depending on the result, the program will go to lines 500, 510, 520, or 530, where the direction of the caterpillar is changed. (Notice that these lines do not *move* the caterpillar; they only change the direction of the move.)

Lines 130-140 place the mushroom. If M is greater than 0, the program creates a mushroom using the routine at line 800. If it is equal to 0, 1 is added to M. Since M is reset to 0 when a mushroom is added, and the program goes back to line 100, this will PRINT a mushroom every other time the program goes through a loop.

A message subroutine in lines 200-300 PRINTs a message (A$) in row (ROW) and column (COL). This is similar to the technique used in earlier games.

Lines 300-397 move the caterpillar in a way different from other games in this book.

Line 307 PRINTs a new body segment at the place where the head was. This is the first step in moving the caterpillar.

Line 310 uses a new command, ON GOTO. This takes a variable, D in this case, and reads what is there. It goes to the first line number it finds after the command if D = 1. That's line 320. If D = 2, the program goes to line 330; if D = 3, then line 340, and so on. D is 1, 2, 3, *or* 4. If it is anything else, the program stops. (If you use ON GOTO, make sure that your variable cannot be greater or less than the number of line numbers you have after GOTO.)

Lines 320-325 decrease the row of the head by 1, lines 330-335 increase the column by 1, lines 340-345 increase the row by 1, and line 350 decreases the column by 1. All except the last line number send the program to line 360, where the next part

of the program continues. The result of these lines, coupled with line 310, is to change the row or column numbers of the head in accordance with the direction variable, D, by using ON GOTO.

Lines 360-370 check to see what is in the new column and row before the head PRINTs. If a space is there, the program goes to line 380. A poison mushroom in the location shifts the program to line 900. If anything *except* an edible mushroom is there, it must be a wall or a caterpillar body part, and the program goes to line 400 where the crash begins.

Line 372 begins the routine that eats the mushroom.

Lines 373-374 displays the new SCORE by using the general message subroutine beginning at line 200.

Line 380 then puts a new head in the proper row and column, while line 381 adds 1 to the K pointer. Lines 382-384 put the new row and column of the head into the proper elements of the arrays R and C.

Line 387 is a complex line that checks to see if the length of the caterpillar is greater than the SCORE + 3. If it is, the program goes to line 390, where the old tail position is erased, causing the illusion that the caterpillar has moved. If you create a new head and erase the old tail, the caterpillar will seem to have moved. You want this to happen except in two situations: You want the caterpillar to get longer when the SCORE is increased (because the caterpillar must grow when it eats), and you want the caterpillar to get longer when you first start the game, so that it will be at least two body segments and one head. At this point, K is one greater than the length of the head and body.

If K is not greater than the SCORE + 3, line 300 PRINTs a new head and body segment, and keeps doing so until the length + 1 is greater than the SCORE + 3.

Line 390 erases the tail.

Lines 391-395 take care of the array. You don't want to keep increasing the length of the array because it will get longer than memory. What happens is that line 391 subtracts 1 from K, and lines 392-395 set up a FOR/NEXT loop which transfers the head and body segment row and column back one element in the two arrays. In other words, the row and column numbers of the second-to-last segment is transferred so it becomes the row and column of the end (because the end was just erased). Since K is one less than before, the K + 1

segment (the head) is transferred and becomes the K segment, which is now the row and column of the head at the *new* K position in the array. This may seem confusing, but it's a valuable technique for keeping track of something like the caterpillar's head and body segment rows and columns. As the caterpillar gets longer, and K is a bigger number, the FOR/NEXT loop is so large that it takes longer to make this transfer, but that's acceptable since a large caterpillar would naturally be sluggish. Whenever you're creating a game, make sure that programming problems fit into the context of the game's world.

Lines 400-420 PRINT the crash message and send the program to line 930 for the ending routine.

Lines 500-535 calculate the caterpillar's direction, based on the joystick movement.

Line 501 makes sure that diagonal movements are not allowed.

Line 502 changes D, the direction variable, to 1, to point the caterpillar up. Line 505 then sends the program back to the main loop.

Similarly, lines 510-515 change the direction to the right; lines 520-525, to down; and lines 530-535 change the direction to the left.

Lines 800-850 put the mushrooms in the garden. Line 810 changes M, the counter, back to 0 for the next time through the loop.

RANDOMIZE is used in line 815 to make sure that RND will be different each time.

Lines 820-825 create a random row and column for the mushroom to appear in, and lines 830-840 check to see if something is already in that row and column. If so, the program goes back to line 820 and tries again, until it finds an empty space for the mushroom. This is important because you don't want mushrooms appearing on top of the walls, caterpillar, or other mushrooms.

Line 842 checks to see if the random number is less than .7. Only then will the program PRINT an edible mushroom. A poison mushroom is put on the screen if RND>.7.

Lines 900-990, the end routine, print the message that you ate a poison mushroom, print your score, and let you have another chance to play the game.

11 Mushrooms

Variables

ROW	Row number of the message.
COL	Column number of the message.
COUNT	Variable used to see how often the caterpillar will move.
R	Array that stores the row numbers of each segment of the caterpillar's body.
C	Array that stores the column numbers of each segment of the caterpillar's body.
CR	Row of the caterpillar's head.
CC	Column of the caterpillar's head.
D	Direction that the caterpillar will travel the next time it's put on the screen.
K	Pointer that shows where the head of the caterpillar is stored in the arrays R and C.
M	Counter used to see how often the mushrooms will be put on the screen.
JR	Joystick Row variable.
JC	Joystick Column variable.
N	Variable that CALL GCHAR uses to store the character value it finds.
MR	Mushroom Row.
MC	Mushroom Column.

Special Notes. The caterpillar in this game is an unusual kind of creature, since it can get longer and longer each time. By using an array, you can keep track of every segment and what row and column it's in. Of course, you have to read just the arrays every time you move, because the row and column of the head and tail change.

Using arrays is often a good technique when you have complex figures on the screen, or you are using changing shapes. Just be sure you don't have an array that gets bigger than your memory can handle.

ON GOTO is a useful command that can help when the program has to make complicated choices. One command can replace, in this case, four commands. (The first would be IF D = 1 THEN 320, the second would be IF D = 2 THEN 330, and so on.)

Often you must use tricky logic to make your program work correctly. An example of this is line 370, which assumes that if the object in the row and column is not an edible mushroom, it must be a body segment or a wall, since everything

else was already tested for in lines 365-367. The logic in line 387 is especially tricky since it is testing for two separate cases, one in which the caterpillar is less than three segments long, and the other if the score has increased since the last time the caterpillar was PRINTed. The use of the variable K in this program is worth studying, because it is used as a pointer to where the head row and column values are in the arrays. K will increase and decrease as the array is first increased and then shortened to accommodate the transfer of all elements backward by one element.

Changing the Game

- You can change the counters in this program to create more or fewer mushrooms each time the main loop is processed. You can also change the counter that moves the caterpillar so that it moves more or less often. If you move it more often, you run the risk of having the joystick's movements not being acknowledged in time and your caterpillar may crash. You can also make it harder by putting in a higher ratio of poison to edible mushrooms.
- You could change this game quite a bit by putting in more walls to make it a maze. If you do that, you could also add creatures that chase the caterpillar, and you could have different kinds of objects for your caterpillar to eat. By using CALL GCHAR, you can create all kinds of obstacle courses for a character to go through, testing each time to see whether a wall or an edible object is there. (See "Hobo Party" in Chapter 12 for an example of a maze-chase game.)
- You could make this game different by setting a top limit on how big the caterpillar can get, so that it won't slow down too much.
- To make this an interesting two-person game, you could have two caterpillars on the screen, and keep them from crashing into each other while they tried to get mushrooms.

You could add sounds for munching the mushrooms, sounds for each time the caterpillar moved, and sounds when it ran into a wall or itself.

Lost in the Garden. To see how easy it is to change the game by adding a few lines, you can create walls inside your garden, and, at the same time, make it easier for the caterpillar by having it grow to a length of only three segments.

11 Mushrooms

Whenever you're changing a game, make sure that it remains balanced. If you make it harder, think about making something else easier, or it will be *too* hard.

Begin by adding:

```
59 GOSUB 3000
```

This will shift the program to line 3000 to place more walls inside the garden. Here's the subroutine which does that:

```
3000 REM PUT DOWN MORE WALLS
3004 RANDOMIZE
3005 LET Z$=""
3010 FOR I=1 TO 4
3020 LET ZR=INT(RND*18)+4
3025 FOR J=1 TO LEN(Z$)
3026 IF CHR$(ZR)=SEG$(Z$,J,1)THEN 3020
3027 NEXT J
3030 LET Z$=Z$&CHR$(ZR)
3040 CALL HCHAR(ZR,INT(RND*10)+2,42,INT(RND*20)+1)
3050 NEXT I
3060 RETURN
```

This subroutine puts down four walls of random length at random places in the garden.

Line 3004 adds RANDOMIZE to make sure that the walls will be put in different places each time.

Z$ is used as a temporary variable to see at which rows the walls have already been put down.

The main FOR/NEXT loop is between lines 3010-3050. Line 3020 sets up a random row number. A secondary FOR/NEXT loop is between lines 3025-3027. Line 3026 tests to see if the row just picked is inside the variable Z$. If it is, you want another row, so the program goes back for another try.

Line 3030 adds the new row to Z$ by using & to add it to the string. Line 3040 puts the wall down at the row that was chosen, starting at a random column and going along for a random length.

To complete the modification, change this line:

```
387 IF K>3 THEN 390
```

By changing this, you make sure that the caterpillar will grow no longer than three segments.

12 Hobo Party

12 Hobo Party

Ever wonder how you can create animation on the TI? "Hobo Party" will show you the principles of animation, creating characters that move their legs and arms. You'll also see how to create mazes, and characters that move in patterns inside the maze.

How to Play

The object of the game is for the hobo to gather all the pans that are scattered throughout the junkyard before the time limit is up.

You're the hobo, and your legs are always moving. You can move through the junkyard by using the joystick. However, you can't cross any junk piles; instead, you must go around them. You'll start out at a different place in the junkyard every time you play.

There's a guard, and his arms are always waving as he holds up his billy clubs. If he gets next to you, the game is over.

Along the bottom row you'll see a colored bar. This shows how much time is left before the party starts. If you haven't gathered up all the pans by then, you've lost the game. The bar gets shorter as time passes.

You pick up a pan by simply moving on top of it. Each pan you get will be transferred to the row immediately above the time bar so you'll know how many pans you've gotten. There are 32 pans you must pick up.

The game starts out by displaying the title and drawing the maze. The maze walls are made up of old boots, tires, TV sets, bottles, and trash bags.

After the maze is on the screen, the 32 pans are put on the screen. Each time you play, the pans are placed in different locations in the maze. Your hobo is then placed in the maze.

The guard is placed in the center of the maze and immediately starts coming after you. If the guard passes over a pan, he moves it out of the way. Sometimes the guard doesn't spot you and will wander around, but sooner or later he'll find you, especially if you move around.

If you get all 32 pans before the time is up, the game's over and you'll have another chance to play. If the guard catches you first, the game ends, you'll find out how many pans you got, and you'll have the option to play again.

12 Hobo Party

Program 12-1. Hobo Party

```
10 REM HOBO PARTY
15 CALL CLEAR
17 RANDOMIZE
20 GOSUB 3000
30 GOSUB 2000
40 GOSUB 4000
100 REM MAIN LOOP
110 CALL CHAR(33,"1C1C083E08142241")
112 CALL CHAR(42,"1C1D0936545C1414")
120 LET COUNT=COUNT+1
122 IF COUNT<20 THEN 130
125 GOSUB 200
130 GOSUB 300
150 CALL CHAR(33,"1C1C083E08141414")
152 CALL CHAR(42,"1C5C4836151D1414")
160 GOSUB 500
190 GOTO 100
200 REM SCORE
205 CALL SOUND(100,440,0)
210 LET COUNT=0
215 LET TIME=TIME-1
217 IF TIME=0 THEN 5000
220 CALL HCHAR(24,1,116,TIME)
225 IF TIME>31 THEN 230
227 CALL HCHAR(24,TIME+1,32)
230 CALL HCHAR(23,1,104,PAN)
290 RETURN
300 REM JOYSTICK
310 CALL JOYST(1,JC,JR)
320 IF JR=4 THEN 410
322 IF JR=-4 THEN 430
324 IF JC=4 THEN 420
326 IF JC=-4 THEN 440
330 RETURN
350 REM CHECK HOBO MOVE
355 CALL GCHAR(RR,CC,C)
360 IF C<>104 THEN 370
363 LET TIME=TIME+1
364 LET PAN=PAN+1
365 CALL SOUND(100,880,0)
366 CALL VCHAR(RR,CC,32)
367 GOSUB 200
368 IF PAN>31 THEN 900
369 GOTO 380
370 IF C=32 THEN 380
375 CALL SOUND(100,110,0)
376 LET RR=HR
377 LET CC=HC
```

```
378 GOTO 330
380 CALL VCHAR(HR,HC,32)
381 CALL VCHAR(RR,CC,33)
382 LET HR=RR
383 LET HC=CC
390 GOTO 330
400 REM MOVES U R D L
410 IF JC<>0 THEN 330
412 LET RR=HR-1
414 GOTO 350
420 IF JR<>0 THEN 330
422 LET CC=HC+1
424 GOTO 350
430 IF JC<>0 THEN 330
432 LET RR=HR+1
434 GOTO 350
440 IF JR<>0 THEN 330
442 LET CC=HC-1
444 GOTO 350
500 REM CHECK GUARD MOVE
502 LET TRY=0
503 LET FLAG=0
505 LET CI=0
506 LET RI=0
510 LET DR=HR-GR
512 LET DC=HC-GC
514 IF ABS(DR)>ABS(DC)THEN 530
515 LET FLAG=1
520 IF DC>0 THEN 525
522 LET CI=-1
523 GOTO 550
525 LET CI=1
527 GOTO 550
530 IF DR>0 THEN 535
532 LET RI=-1
533 GOTO 550
535 LET RI=1
550 CALL GCHAR(GR+RI,GC+CI,C)
552 IF C=33 THEN 700
555 IF C<>104 THEN 560
557 LET STORE=104
558 GOTO 600
560 IF C<>32 THEN 580
565 LET STORE=32
566 GOTO 600
570 LET RI=INT(RND*3)-1
572 LET CI=INT(RND*3)-1
575 GOTO 550
580 IF TRY=1 THEN 570
```

12 Hobo Party

```
581 LET TRY=1
582 IF FLAG=1 THEN 530
583 GOTO 520
600 REM MOVE GUARD
610 CALL VCHAR(GR,GC,STORE)
620 CALL VCHAR(GR+RI,GC+CI,42)
630 LET GR=GR+RI
640 LET GC=GC+CI
650 RETURN
700 REM GUARD CATCHES HOBO
705 CALL CLEAR
710 PRINT "THE GUARD GOT YOU"
800 REM FINAL MESSAGE
810 PRINT
820 PRINT "YOU GOT ";PAN;" PANS"
830 PRINT
840 PRINT "WANT TO PLAY AGAIN?"
850 PRINT "PRESS Y FOR YES"
860 CALL KEY(3,K,S)
870 IF S=0 THEN 860
875 IF K=89 THEN 10
880 END
900 REM ENOUGH PANS
905 CALL CLEAR
910 PRINT "YOU GOT ENOUGH PANS!"
920 GOTO 800
2000 REM MAZE GENERATOR
2050 CALL CLEAR
2100 LET A$="HOBO PARTY"
2110 LET ROW=1
2120 LET COL=11
2130 GOSUB 2700
2160 FOR II=1 TO 4
2165 CALL VCHAR(1,2*II+2,33)
2170 CALL VCHAR(1,2*II+21,33)
2180 NEXT II
2190 RESTORE
2200 REM PUT MAZE ON SCREEN
2205 RANDOMIZE
2210 FOR J=3 TO 22
2220 READ A$
2230 FOR I=1 TO 32
2240 IF SEG$(A$,I,1)<>"0" THEN 2260
2250 LET EL=32
2255 GOTO 2270
2260 LET EL=INT(RND*5)+96
2270 CALL VCHAR(J,I,EL)
2280 NEXT I
2290 NEXT J
```

```
2300 REM PUT PANS ON SCREEN
2310 FOR I=1 TO 32
2320 LET PR=INT(RND*18)+4
2325 LET PC=INT(RND*30)+2
2330 CALL GCHAR(PR,PC,C)
2335 IF C<>32 THEN 2320
2340 CALL VCHAR(PR,PC,104)
2350 NEXT I
2400 RETURN
2500 REM MAZE DATA
2510 DATA 001111111111111111110001111111100
2511 DATA 011000000000000000111110001111111
2512 DATA 110000000000000000000010000000001
2513 DATA 110011111111111100000000000000011
2514 DATA 100000110000000000000000011110010
2515 DATA 100000000000000000011001100000001
2516 DATA 101000000011111100111111000000001
2517 DATA 111111000000000111100000000001001
2518 DATA 011000000000000010000000011001
2519 DATA 010000011110000011000111111100001
2520 DATA 110001111000010011000100000000011
2521 DATA 100000000000010001000000000000110
2522 DATA 100000000000010001000000110000010
2523 DATA 100111111100010001001000100000001
2524 DATA 100000000000110001001000100000001
2525 DATA 100000000000100000001000100001001
2526 DATA 110000011111000000001100100001001
2527 DATA 111000000000000100011100000001011
2528 DATA 011110000000001110001110000011110
2529 DATA 000111111111111001111101111111000
2700 REM PRINT MESSAGE
2710 FOR I=1 TO LEN(A$)
2720 CALL VCHAR(ROW,COL+I,ASC(SEG$(A$,I,1)))
2730 NEXT I
2740 RETURN
3000 REM CHARACTERS
3010 REM TRASH
3020 CALL CHAR(96,"28103C7EFFFFFF7E")
3030 CALL CHAR(97,"3C7EE7C3C3E77E3C")
3040 CALL CHAR(98,"1408FF85878587FF")
3050 CALL CHAR(99,"000F0F0F0F3F7F77")
3060 CALL CHAR(100,"44444EEEEEEEEEEE")
3070 REM PANS
3075 CALL CHAR(104,"0000003E38380000")
3100 CALL SCREEN(4)
3110 CALL COLOR(9,13,1)
3120 CALL COLOR(10,9,1)
3200 RETURN
4000 REM INITIAL VARIABLES
```

12 Hobo Party

```
4005 REM HOBO
4010 LET HR=INT(RND*18)+4
4015 LET RR=HR
4020 LET HC=INT(RND*30)+3
4025 LET CC=HC
4030 CALL CHAR(33,"1C1C083E08141414")
4035 CALL COLOR(1,2,1)
4040 CALL GCHAR(HR,HC,C)
4050 IF C<>32 THEN 4010
4060 CALL VCHAR(HR,HC,33)
4100 REM GUARD
4105 LET GR=10
4106 LET GC=18
4130 CALL CHAR(42,"1C1D0936545C1414")
4135 CALL COLOR(2,2,1)
4160 CALL VCHAR(GR,GC,42)
4200 LET COUNT=0
4210 LET TIME=33
4220 CALL COLOR(11,9,1)
4225 CALL CHAR(116,"CCCCCCCCCCCCCC")
4230 LET PAN=0
4300 GOSUB 200
4310 LET STORE=32
4900 RETURN
5000 REM TIME IS UP
5010 CALL CLEAR
5020 PRINT "YOUR TIME IS UP"
5030 GOTO 800
```

Program Structure

Line 10 is the title, line 15 clears the screen, and line 17 uses the RANDOMIZE command.

Line 20 calls a subroutine which starts in line 3000. This subroutine creates the characters and colors needed to create the maze walls and pans. Since this subroutine and the next (which creates the maze) take up a lot of space in the program, it's better to put it in a subroutine nearer the end of the program so that it will be out of the way. This speeds program flow because the less you have at the beginning of a program, the faster the TI finds things, since it starts looking for certain things at the lowest line numbers.

Line 30 calls a subroutine that actually creates the maze, starting in line 2000.

Line 40 calls a subroutine that starts in line 4000 which sets up all the initial conditions of the rest of the game.

Line 100 starts the main loop.

Lines 110-112 use CALL CHAR to give a shape to both the hobo and the guard. These shapes will be used to animate the hobo and guard. (See lines 150-152 for more explanation.)

Lines 120-125 use the variable COUNT to keep track of how much time passes in the game. Each time the program goes through the main loop, COUNT is increased by 1. If COUNT is less than 20, the program goes to line 130 and the main loop continues. If COUNT equals 20, the program goes to line 125. This sends the program to a subroutine in line 200 which changes the time display so the player knows time is passing. (Notice that by using a jump to line 130, line 125 can call a subroutine. This is needed because other parts of the program will use the subroutine in line 200. Whenever possible, use subroutines, for then they can be easily used again and again.)

Line 130 sends the program to a subroutine starting in line 300, a joystick reading subroutine.

Lines 150-152 use CALL CHAR to give *different* shapes to the guard and hobo. Compare these shapes to the CALL CHAR commands in lines 110-112. They are similar, but the two shapes for the hobo make the hobo's legs seem to move, and the two shapes for the guard make his arms seem to move. This is one way to create animation on the TI. All that's been done is to use CALL CHAR to create two shapes, but because they change every time the program goes through the loop, the characters seem to move. A simple addition like this can make a game more fun, even though it doesn't affect game play.

Line 160 makes the program go to a subroutine that starts in line 500. This subroutine makes the guard move in the direction of the hobo.

Line 190 ends the main loop.

Line 200 begins the subroutine that puts the time bar chart on the bottom of the screen, and puts the number of pans that the player has obtained on the row above the bar.

Line 205 uses CALL SOUND to let the player know that the time and/or number of pans is being changed. This is helpful since the player will be watching the center of the screen most of the time.

Line 210 resets COUNT to 0 so that when the program goes back to the main loop, it can count to 20 again.

Line 215 subtracts 1 from the variable TIME, which is used to keep track of remaining time.

Line 217 checks to see if time has run out. If TIME is 0, the program jumps to line 5000 and ends the game.

Line 220 puts the bar chart on the screen. By using CALL HCHAR, character 116 is PRINTed the number of time to equal the value of variable TIME. TIME starts at 32 and decreases by 1 each time this subroutine is called.

Lines 225-227 put a space (character 32) after the time bar chart's end, so that the bar gets shorter. Line 225 checks to see if the bar chart is 32, which it will be the first time through. If it is, the program does not put a space after the bar chart's end; if the TIME variable is less than 32, line 227 puts a space at the column that is one more than the length of the bar chart (measured by TIME). Without these two lines, you wouldn't know that the bar is getting shorter.

Line 230 puts the number of pans that the player has gotten on the screen. By using CALL HCHAR with character 104, which is the pan character, the number of pans PRINTed equals PAN. If PAN is 0, nothing is put on the screen.

Line 300 begins the subroutine that reads the joystick.

Lines 310-326 are the same as in other games in this book, using CALL JOYST to see if the joystick has been moved, and then sending the program to specific lines depending on the joystick direction.

Lines 350-390 process the hobo's intended move and see whether the hobo can actually move.

Line 355 uses CALL GCHAR to see if the new move will work. Whatever is in C is the object that is currently in the location where you want the hobo to go, as determined by the new row and column, RR and CC.

Line 360 tests to see if C is not equal to 104. If it isn't, that means the place you want to move the hobo to *doesn't* contain a pan, and the program goes to line 370.

If there is *is* a pan in the way, lines 363-369 take care of what happens, because you'll be picking up that pan. Line 363 increases TIME by 1. The time will not actually increase, but because the subroutine that puts the pans taken on the screen also decreases TIME, you must *increase* it here so that there will be no change in the TIME variable. Line 364 increases PAN. Line 365 uses CALL SOUND to let the player know a pan has been picked up. Line 366 erases the pan. Line 367 uses the

subroutine in line 200 to put the new number of pans on the screen. Line 368 checks to see if all the pans have been picked up. If they have, the program goes to line 900, where the program ends. Line 369 sends the program to line 380, where the hobo is put on the screen at its new position.

If there is *no* pan in the way, line 370 tests to see if a space is present. If it is, all is well, and the program goes to line 380.

If there is something else in the way, it must be one of the junkyard walls or maybe the guard. If either is true, lines 375-378 make sure that the hobo can't go in that direction.

Lines 375-376 reset the new row and coumn variables so that they are the same as the old row and column variables.

If all is well, lines 380-390 put the hobo at the new row and column.

Line 380 uses CALL VCHAR to erase the hobo at the hobo's old row and column, HR and HC, and line 381 uses CALL VCHAR to put the hobo at the new row column, RR and RC.

Lines 382-383 change the old row and column to the new row and column, so that the next time through the loop, the old becomes the new.

Line 390 sends the program to line 330, which is a RETURN to the main loop. A RETURN *could* have been put here, but by sending all the branches of a complicated subroutine like this to a central RETURN line number, you can see the program flow easier, and you don't risk having a RETURN in the wrong place.

First, line 410 checks to see if the joystick column variable was 0. If it wasn't, the joystick was moved diagonally, and the program goes to line 330, which returns to the main loop. (You should always check for this kind of input problem to make sure that what the player wants actually happens. It can be very frustrating for the player to think that he or she is moving the joystick up and have the character move sideways because there was a slight diagonal tug on the joystick.)

Line 412 creates a variable RR, which is used for the new row you want the hobo to move to. You don't want to actually change the row yet because the hobo might not be able to go in that direction.

Finally, line 414 sends the program to line 350, where the hobo's move is processed.

Similarly, lines 420-444 test the joystick's input for diagonals,

create a new row or column variable that will be used to see if the hobo can go in that direction, and shift the program to line 350.

Lines 500-650 move the guard. Every time the program goes through the main loop, the guard moves. Lines 500-575 calculate the guard's move, and lines 600-650 move the guard.

Lines 502-503 create variables that will be used as flags to test which way the logic will flow in this subroutine. TRY is 0 and will be used to see if one logic path was followed, and FLAG will be used to see which way a different logic path went.

Lines 505-506 set the variables CI and RI to 0. These will be used for the increase (or decrease) of the guard's row or column.

Lines 510-512 create new variables which will be used to calculate the *difference* between the row and column positions of the hobo and guard.

Line 514 takes the absolute values of the *differences* between the rows and columns of the hobo and guard, and sees which is greater. You want to determine which way the guard will move first. You've already calculated the differences between the rows and columns, and you want to go in a pattern that will use the shortest path, which will be the difference that is *greatest*. In other words, if the distance between the rows is greater than the distance between the columns, you want to decrease that distance first. If the row difference is greater than the column difference, the program will go to line 530; if the column distance is greater than the row distance, the program will go to line 520. Notice that the absolute values were taken with ABS. This was used because you're concerned only with the numerical distance. If you don't take the *absolute* value, you might get wrong answers because the difference may be a minus number if the hobo is, for example, to the left of the guard.

Lines 515-527 are used for the guard's move calculation if the column difference is to be decreased.

First of all, line 515 sets FLAG to 1. Line 520 tests to see whether the difference between columns is greater than 0. If it is, the hobo is to the left of the guard, and the program goes to line 525. If it's not greater, the hobo is to the right of the guard, and the program goes to line 522.

Line 522 sets CI to −1. This will be used later to move the

guard's position to the left.

Lines 525-527 are used if the hobo is to the left of the guard. CI = 1 is used later to move the guard right.

Lines 530-535 operate the same way as lines 515-527, except that the hobo is above or below the guard, and CR is changed to 1 or −1 to move the guard down or up, depending on whether the hobo is below or above the guard. The only difference is that FLAG is not changed, so that it will remain 0. The FLAG variable will be used later to tell whether the guard *attempted to move* left or right (FLAG = 1) *or* up or down (FLAG = 0). Remember, at this point in the program, you are calculating only where the guard will *attempt* to move.

Line 550 uses CALL GCHAR to see where the attempted move will go. The character in the row and column where the guard wants to go is stored in the variable C.

Line 552 tests to see if C is 33. If it is, that means that the guard has caught the hobo, and the program goes to line 700.

Line 555 tests to see if the attempted move is *not* equal to character 104, which is the pan character. You must be careful that the guard does not accidentally erase a pan. If no pan is in the way, the program goes to line 560.

If a pan is in the way, the guard can move there. First, line 557 puts the number 104 into a variable called STORE. In the initialization part of the program, STORE was set to 32. STORE is used to leave behind either a space or a pan when the guard moves. If a guard is about to move on top of a pan, STORE will be set to 104 so that when it moves, the pan will be put where the guard moved from. The program then goes to line 600 to move the guard.

Line 560 tests to see if the position the guard moved to is *not* equal to 32. If it is not, a maze wall is blocking the way, and the program goes to line 580.

Line 565 makes sure that STORE contains a space (ASCII 32) so that when the guard is put in its new position, a space will be put down in the old position. This is necessary to prevent another pan from being PRINTed.

Line 566 sends the program to line 600 so that the guard can be PRINTed.

Lines 570-575 are accessed if the first and second attempts to move the guard failed. Lines 570-572 generate random increases or decreases for the guard's row and column, and line 575 sends the program back to line 550 to test these new

values. If they don't work, lines 570-575 repeat again and again until a path for the guard's movement is found. When one is found, the guard is moved by line 600.

Lines 580-583 determine the flow of the logic in the guard's attempts. The first time through the guard's subroutine, the program will attempt to move the guard in the row or column direction that is still toward the hobo, but which is the greatest *difference* in row or column direction. However, if this direction didn't work because something is in the way, the program will next try to go in another direction toward the hobo. Here's an example: If the guard is in the middle and the hobo is to the right or left, and the row difference is greater than the column difference, the guard will first attempt to move toward the row that is closer to the hobo. If the way is blocked, the guard will next try to move toward the column that is closer to the hobo. Finally, if neither of these ways works, a random move away from the hobo takes place. This may help the guard escape from a blind spot he may be trapped in.

Line 580 sees if TRY is equal to 1. If it isn't, this is the first time through the testing procedure. The first attempt has failed, and now a second attempt is made.

Line 581 sets the TRY variable to 1, so that the next time the program reaches this point, if the second attempt fails, line 580 sends the program for a third attempt.

Line 582 sees if the FLAG variable was set to 1. If it was, the first try was in a column direction, so the program goes back to line 530 and tries to see if moving in a row direction will help. If FLAG was 0, the first attempt was in a row direction, so line 583 makes the program go back to line 520 to see if moving in a column direction will help.

Lines 600-650 actually move the guard. Line 610 uses CALL VCHAR to put a space where the old row and column of the guard was. A space is put down if STORE is equal to 32, but a pan is placed if STORE is equal to 104. Line 620 uses CALL VCHAR to put the guard in the new row and column, as calculated by GR+RI and GC+CI. Finally, lines 630-640 change the old guard row and column to the new row and column.

Lines 700-710 are accessed if the guard catches the hobo. Line 705 clears the screen, and line 710 prints a message.

Lines 800-880 are the end of the program. This will be gone to when the guard catches the hobo, the time runs out, or

the hobo gets all the pans. This is programmed the same way as the endings of other games in this book, to give the final score, a chance to play again, and end the program.

Lines 900-920 are used if the hobo got all the pans. Line 905 clears the screen, line 910 prints the message, and line 920 sends the program to line 800 for the ending.

Lines 2000-2740 contain the subroutine that generates the maze and puts the pans and the title on the screen.

Lines 2100-2130 set up variables which use the subroutine starting at line 2700. The subroutine at line 2700 puts the title message on the screen, HOBO PARTY, PRINTed at row 1, column 11.

Lines 2160-2180 put the exclamations on either side of the title. Dancing hobos appear from these characters when the program is finally set up. The exclamation marks are put on the screen using CALL VCHAR and a FOR/NEXT loop.

Line 2190 is the word RESTORE. This is a very important safety measure when you're using DATA statements, such as those which generate the maze. RESTORE makes sure that when READ reads a DATA statement, it starts with the very first statement.

Line 2200 begins the actual maze generation.

Line 2210 starts a FOR/NEXT loop that is used to put each row of the maze on the screen. The first row is 3 and the last row 22; J is used for the row number.

Line 2220 READs the first DATA statement it finds. The DATA statements in this program are in lines 2500-2529. Each DATA statement is a collection of 32 numbers read as one string, A$. The numbers are either 0 or 1; a 0 indicates a space and a 1 indicates that a character is to appear on the screen. The maze generator is set up this way so that the programmer can easily make changes in the layout of the maze, by merely changing the 0's or 1's. There are 20 DATA statements, each representing one row.

Line 2230 starts a second FOR/NEXT loop which is used to translate the 0's and 1's into specific objects to create the maze. Variable 1 is used for the counter here and is also the column number.

Line 2240 looks at each segment of the A$ variable. If the segment is not equal to 0, the program will go to line 2260. Notice that 0 is used because 0 is a string, not a numeric variable. If the segment is a 0, the program goes to line 2250.

Line 2250 sets the variable EL (ELement) to 32. EL is later used to decide what is put on the screen. Since the segment was 0, EL will be 32 so that a space can be printed.

Line 2255 then sends the program to line 2270 so that the space can be put on the screen.

Line 2260 is accessed if the segment of A$ is equal to something besides 0 (in other words, if the segment is 1). This line generates a random number, stored in EL, between 96 and 100. EL is the character printed. Trash bags, tires, TV sets, boots, and bottles are the custom characters defined in the subroutine at line 3000. Because EL is chosen randomly each time through the loop, the maze consists of walls of varying objects. The overall shape is the same, but the texture is different each time because of what makes up the walls.

Line 2270 uses CALL VCHAR to put a character on the screen in row J and column I.

Lines 2280-2290 close off the FOR/NEXT loops. When you have one loop inside another, make sure that you have them closed correctly. If the outer loop is J, you should put NEXT J last. When the program has gone through all of the J and I loop combinations, the maze is on the screen.

Lines 2300-2350 put the pans on the screen. Line 2310 sets up a FOR/NEXT loop that will put 32 pans on the screen. Lines 2320-2325 create a random row and column for the pan. Notice that no number will be generated that will be outside the maze boundaries. Line 2330 uses CALL GCHAR to see if the row and column generated are already occupied. If they are not equal to 32 (a space), the program goes back to line 2320 to try again. Line 2340 is gone to if a space *was* at the row and column numbers that were generated. CALL VCHAR puts a pan (character 104) in the row and column selected.

Lines 2500-2529 are the DATA statements used to create the maze. (Notice that these DATA statements are not part of the program, in the sense that no GOTOs or GOSUBs go to these lines. They could have been put in any part of the program that is not in the program flow. When a READ statement is used, the program starts at the beginning and looks for the first DATA statement it finds. RESTORE must be used to make sure that the first DATA statement is used; otherwise, there is a chance that if you tried to reuse the DATA statements, they would start *after* the last DATA statement.)

Lines 2700-2740 contain a subroutine which puts a

message on the screen. This is similar to message subroutines used in other programs in this book. CALL VCHAR is used to take a string, A$, and by using ASC(SEG$), puts each character of the string in the proper row and column.

In lines 3000-3200 is a subroutine which creates the character shapes and colors of the maze and pans.

Lines 3020-3060 use CALL CHAR to create the shapes of the trash bags, tires, TV sets, boots, and bottles.

Line 3075 creates the shape of the pan using CALL CHAR.

Line 3100 colors the screen light green.

Line 3110 colors the maze objects dark green. Dark green is used with light green so that the shapes of the objects will show through. If you use colors that contrast too much, some of the dots that make up the objects will seem to blend into the background. Line 3120 colors the pans medium red.

Lines 4000-4900 set up the initial variables that start the game.

Lines 4005-4060 put the hobo on the screen. Line 4010 sets the hobo's beginning row to a random row number, HR. Line 4015 sets the new hobo row number variable, RR, to the same value as HR.

Lines 4020-4025 create the old and new hobo column number variables, in a manner similar to lines 4010-4015.

Line 4030 uses CALL CHAR to create the initial shape of the hobo. This shape changes to produce animation in the main loop.

Line 4035 colors the hobo black. Black is used for both the hobo and the guard for the best visibility. You can tell them apart because they have different shapes which are animated to move the hobo's legs and the guard's arms.

Line 4040 uses CALL GCHAR to see what is in the row and column generated for the hobo's location. If the row and column do not have a space (character 32), the program goes back to line 4010 to try again. If the row and column *do* have a space, line 4060 uses CALL VCHAR to put the hobo on the screen.

Lines 4100-4160 put the guard on the screen.

Lines 4105-4106 create the guard's beginning row and column position, in a central location, in row 10 and column 18. Line 4130 creates the guard's beginning shape, which is animated in the main loop. Line 4135 colors the guard black, and line 4160 puts the guard on the screen using CALL VCHAR.

Line 4200 sets COUNT to 0. COUNT is used in the main loop to keep track of time.

Line 4210 sets TIME to 33. TIME is set equal to 33 because the subroutine that uses TIME starts out by subtracting 1. Remember that no more than 32 blocks can be on one line at a time.

Line 4220 sets the color of the blocks that show how much time has elapsed.

Line 4225 uses CALL CHAR to create the shape of one block. The total number of blocks indicates how much time is left.

Line 4230 sets the variable PAN equal to 0. Line 4300 uses the subroutine starting at line 200 to put the number of pans on the screen and also to show how much time has elapsed.

Line 4310 sets the variable STORE equal to 32. STORE is used by the subroutine that moves the guard. Lines 5000-5030 are gone to when the time is up, and the player has not gotten enough pans. Line 5010 clears the screen, line 5020 prints the message that the time is up, and line 5030 sends the program to line 800, where the ending takes place.

Variables

HR	Hobo's current row.
RR	Hobo's attempted row when it tries to move.
HC	Hobo's current column.
CC	Hobo's attempted column when it tries to move.
GR	Guard's current row.
GC	Guard's current column.
COUNT	Variable used to keep track of time in the main loop.
TIME	Variable used to put the amount of remaining time on the screen.
PAN	Number of pans that the player has picked up.
STORE	Variable used to see whether the guard is about to move on top of a pan or space.
A$	String variable which holds a message to be put on the screen *or* a temporary variable for a DATA statement.
ROW	Row that the message will be PRINTed to.
COL	Column that the message will be PRINTed to.
EL	What is put on the screen when the maze is generated.

PR	Pan row.
PC	Pan column.
FLAG	Variable to see whether the first attempt of the guard's move was in a row or column direction.
TRY	Variable to see if the second attempt of the guard's move also failed.
CI	Increase or decrease of the guard's column position move attempt.
RI	Increase or decrease of the guard's row position move attempt.
DR	Difference between the hobo row and the guard row.
DC	Difference between the hobo column and the guard column.

Special Notes. By changing the value that CALL CHAR uses in the main loop, this program is able to animate the two action figures. Even if they are standing still, they still move, because arms wave and legs jump. Using this kind of simple animation can make a game more fun because players expect realistic character movement in arcade games.

The maze generated in this game is a fixed maze, in the sense that the pattern will always be the same. It feels different each time you play because the walls of the game are different. Reading in the shape from DATA statements makes the maze itself easy to modify.

This game shows how to display scores by using a bar chart instead of just putting the number on the screen. Often it is easier for a player to just glance at a bar chart and understand its meaning than it is to see a number. Also, by using the same technique, but with the shape of a pan instead of a block, you can show how many pans have been picked up by the hobo.

Study the logic of how the guard moves in lines 500-560. In this game the idea is to make the guard as smart as possible, so it will be able to catch the hobo. By anticipating all the possible moves the guard can make, you can decide which moves are best. If he can't make one move, you can see which is the best. However, it's important to give the guard a way to move, even if it is not the best way; otherwise, the guard may get stuck, or worse yet, the program may become stuck in a loop, always trying the best way.

Another interesting feature of this game is what happens

when the guard moves over a pan. When you have objects scattered around a screen that you don't want accidentally erased, you must have a way to restore whatever is moved over. This is a powerful technique. By using it, you could create a detailed map and have objects move over the map and store what they move over in a variable, leaving behind what they just went over.

Changing the Game

Besides the usual changes of color, shape, and sound, you can do a lot to change this game.

- The most obvious is to create mazes of different shapes. You can create your own patterns, and enter them as DATA statements.
- You can alter the maze each game by changing into a space one of the five objects that make up the maze walls. For example, if you make the trash bag into a space, the walls will have different shapes. However, if you do this, take care that the outside walls contain no spaces, or the moving characters may be able to escape.
- You can make the game more difficult by letting the guard move more often, or you can make the time shorter. If you want to make the game more difficult, you can create a way for the guard to blast holes in the maze walls so that he can more easily get to the hobo.
- To make it easier, you could add an escape option: If the joystick button is pressed, the hobo could be picked up and carried to a new unknown position on the screen, as a way to escape when the guard is closing in.
- Also, a second or third guard could be added, just by duplicating the guard move routine for a second or third guard.
- You could also create special items, like coffee pots. If the hobo gets these special items, he can chase the guard for a while. This will require the guard to do the opposite of what he did before, and will make the game very complex.
- Finally, you can create a series of mazes for the hobo and guard to go through, and add things like traps, dogs, and so on.

13 Moneybags

13 Moneybags

Using the TI Extended BASIC cartridge's capability to create sprites, you can easily create and move characters on the screen. (If you don't have the Extended BASIC cartridge, you can't use the commands in this game.)

Sprites have the ability to move independently, can be larger than normal characters, can move without erasing anything created on the screen by normal characters, and can detect collisions between one sprite and another.

"Moneybags" is a game that uses 14 sprites: 12 cars that run on a freeway, one money truck that has a bad habit of dropping money bags, and a player figure trying to pick up the money without being run over by the cars.

Before you see how the game works, a short summary of how sprites work on the TI is in order.

Sprites

First of all, you must have the TI Extended BASIC cartridge plugged into your machine. Then, when you power up, and after pressing any key, you'll be given an option between TI BASIC and TI EXTENDED BASIC. Press the 2 key to use TI EXTENDED BASIC.

Most of the differences between Extended and TI BASIC won't be covered here. However, if you want to edit a line, instead of typing EDIT 10, you must type the line number you wish to edit and then press FCTN and the E key simultaneously. Also, if you want to use PRINT to scroll the screen, make sure you don't have anything on line 24 (see the discussion of scrolling in Chapter 9).

Also, you can combine more than one statement on a line by putting two colons (::) between statements.

You can also see how much memory you have left by typing SIZE.

Finally, Extended BASIC allows you to do many other complicated things, such as a more generalized IF/THEN command, using a GOSUB after THEN, or having a simple command follow the THEN.

To use sprites effectively, here's a summary of the commands that affect sprites.

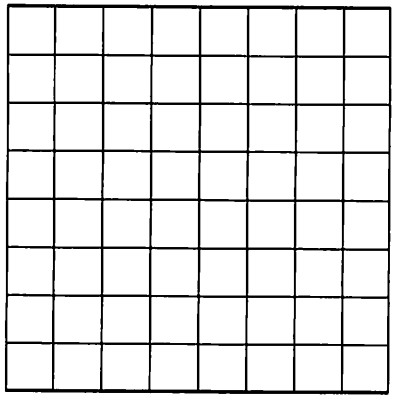
13 Moneybags

CALL MAGNIFY. To use sprites, you must first decide how large you want them. CALL MAGNIFY will make all the sprites a particular size.

CALL MAGNIFY (1) makes sprites the same size as characters, 8 dots wide by 8 dots high. Each sprite is made up of one character, much the same way that a custom character is made. CALL CHAR is used to create the sprite's shape.

Take a look at Figure 13-1 for a moment. Similar to a custom character, a sprite can be drawn using an 8 by 8 grid.

Figure 13-1. Sprite Grid



To draw the sprite, simply fill in the boxes representing the *on* dots, then calculate the hexadecimal code values for each four-dot pattern. Refer to Figure 3-3, Character Combinations and Codes, for the 16 patterns and their code values.

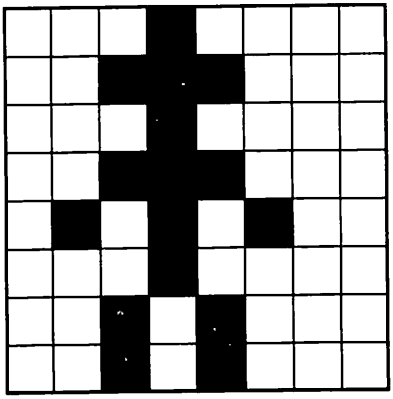
An example of a sprite, already drawn and with its hexadecimal values calculated, is in Figure 13-2.

In fact, this sprite picture is used in Moneybags.

CALL MAGNIFY (2) is the same as CALL MAGNIFY (1) except that the dots have been enlarged, so that they cover more area.

CALL MAGNIFY (3) creates sprites that are 16 dots across and 16 dots down. In other words, each sprite is actually made up of four characters in a square, two characters by two characters. In fact, each CALL MAGNIFY (3) sprite is created from

Figure 13-2. Bank Guard



Code Values

10
38
10
38
54
10
28
28

four characters in sequence, such as character numbers 32, 33, 34, 35. The first character is put in the upper-left corner, the second in the lower-left corner, the third in the upper-right corner, and the fourth in the lower-right corner. When you're choosing four characters to make a sprite like this, pick four that start at a number divisible by 4, such as 32, 36, 40, 44, and so on.

A sprite drawn when CALL MAGNIFY (3) is used would be created on a 16 by 16 dot grid, similar to Figure 13-3. Notice that it's actually four character grids put together.

You can use all four character grids if you want, or you can use only one, two, or three of them. Unused grids will show as blanks, or empty. You can use CALL MAGNIFY (3) to create sprites that are long and thin, or tall and narrow, by using only two of the grids, leaving the other two blank. Moneybags does this creating a sprite like the one in Figure 13-4.
Four characters are needed to draw this sprite, but the characters $n + 1$ and $n + 3$ were left blank. This way the car appears twice as long as it is high.

CALL MAGNIFY (4) is the same as CALL MAGNIFY (3) except that each dot has been enlarged. This is similar to the difference between CALL MAGNIFY (1) and CALL MAGNIFY (2).

Figure 13-3. CALL MAGNIFY (3) Grid

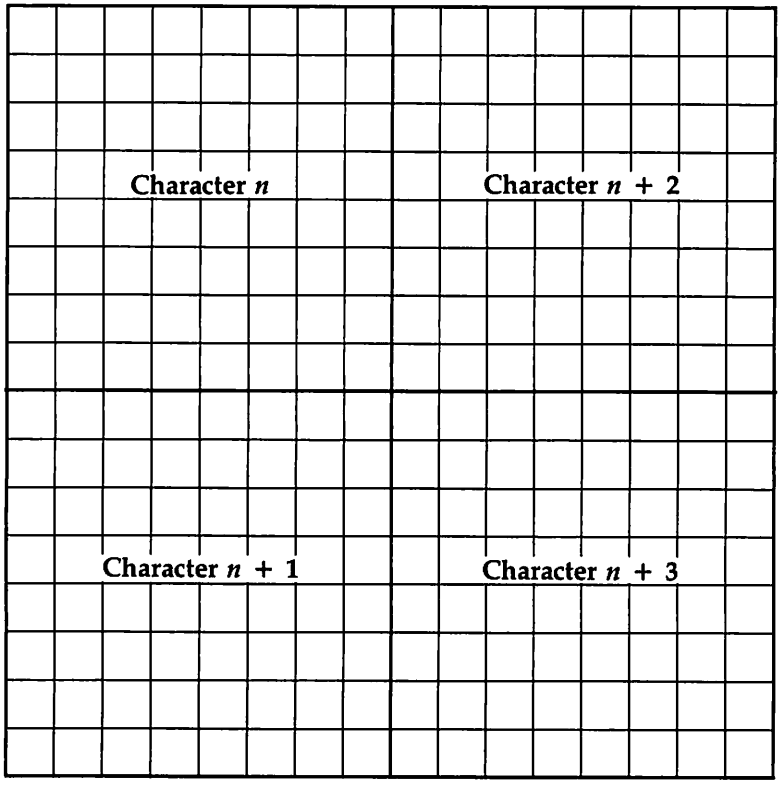
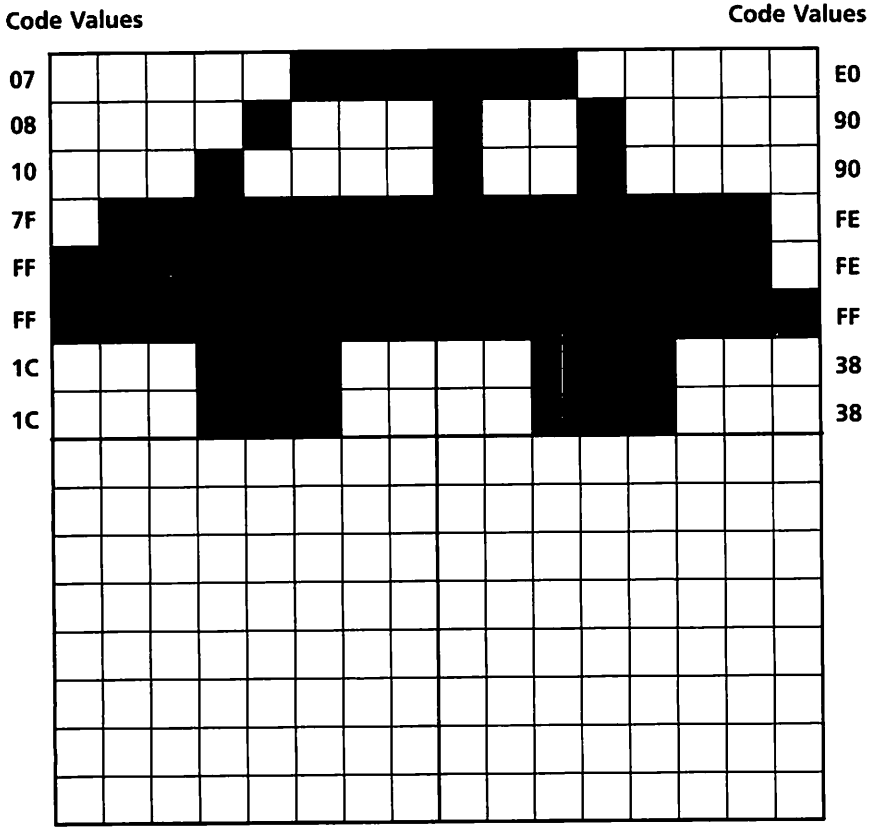


Figure 13-4. Car Sprite



13 Moneybags

CALL CHAR. Once you have decided what CALL MAGNIFY you want to use, create your sprite shapes by using CALL CHAR to define them. This works the same way as CALL CHAR did in creating custom characters. If you are using a CALL MAGNIFY of 3 or 4, you must make sure to put blanks in the characters that you don't want to appear on the screen. You can do this by simply saying CALL CHAR (*n* " "), which will put all zeros in character *n*.

CALL SPRITE. CALL SPRITE puts your sprite on the screen. You can have up to 28 different sprites on the screen at one time. Each sprite has a number (which you must put a # sign in front of).

Each time you create a sprite, you must give it several numbers:

- The first is the sprite number. Between 1 and 28, it must have a # sign in front of it.
- The next value is the character number. If you have a CALL MAGNIFY of 1 or 2, the number you put in will be the ASCII number of the character you've defined, from 32 to 159.

  If you have a CALL MAGNIFY of 3 or 4, you must put in the first ASCII number of the first character, and the TI will use the following three numbers to define the rest of the sprite's shape.
- Next is the number to color the sprite. Any number between 1 and 16 can be used.
- The *dot-row* number is next. Sprites use a different system of rows and columns than normal characters. The TI uses 256 dot-rows, from 1 at the top of the screen down to the bottom. However, the bottom of the screen is at dot-row 192, so any dot-rows beyond 192 are off the bottom of the screen. These extra dot-rows can be useful for hiding sprites temporarily.

  Eight dot-rows are equal to one normal row, so you can calculate a sprite's position by multiplying a normal row by 8 and subtracting 7. For example, dot-row 17 will be at the top of normal row 3. Of course, there is the possibility of overlapping, so you must calculate carefully.
- Next is the *dot-column* value. This is similar to the dot-row numbers; there are 256 dot-columns, starting with 1 on the left side and ending at 256 on the right.

- The sixth value is the *row velocity,* which causes the sprite to move in a vertical direction. The speed depends on the number. If you use positive numbers, the sprites move down, and if you use negative numbers, they move up. If the row velocity is 0, the sprites won't move. Row velocity values are from - 128 to 127, with the higher numbers being quite fast. If the sprite goes off the top or bottom of the screen, it will wrap around, coming back onto the screen, moving in the same direction.
- Finally, you need a value for *column velocity.* This works the same as the row velocity, except that the movement is horizontal. If the numbers are positive, the sprite will move from left to right, and if the numbers are negative, it will move from right to left. If the column velocity is 0, the sprite will remain stationary. You can leave off the row and column velocities and the sprite won't move, but it's a good idea to define them both as 0 just to make sure.

Here's the form for CALL SPRITE:

CALL SPRITE *(#sprite number, character number, color number, dot-row, dot-column, row velocity, column velocity)*

Once you have set up a sprite, it will keep moving until the program tells it to do something else.

If you have *both* the row velocity and column velocity numbers not equal to zero, the sprite will move diagonally.

CALL COINC. Once you have your sprites moving, it's often useful to detect collisions with another sprite.

There are several different ways to use CALL COINC (which stands for Coincide). We'll use it to detect collisions between two particular sprites.

CALL COINC needs three numbers and a variable to make it work:

- The first number is the sprite number you want to test. You must precede it by a # sign.
- The second value is the sprite number you want to check to see if it has collided with the first. You must also precede it by a # sign.
- The third number is called the *tolerance.* This is the distance between the upper-left corners of the two sprites in question. If you have a tolerance of 1, the program will detect a collision only if the sprites' upper-

left corners are in the same dot-row and dot-column. If the tolerance is 8, the upper-left corners have to come only within eight dot-rows and dot-columns of each other.

- Finally, a variable must be inserted that contains a number to tell you whether the two sprites collided or not. The variable contains a 0 with no collision, and −1 with a collision.

CALL COINC's form looks like this:

CALL COINC *(#sprite number, #sprite number, tolerance, variable)*

You can also use CALL COINC to detect if *any* sprites collide or if a sprite collides with a specific dot-row and dot-column.

It's important to note that CALL COINC tells you only whether there is a collision at the time the command is processed. To use it properly, you should have it in the main loop and test it frequently. Sometimes, if a sprite is moving fast enough, a CALL COINC won't happen at the moment of the collision and you'll get a false reading.

CALL LOCATE. If you want to change a sprite's position on the screen, you can use CALL LOCATE to move it to a new dot-row and dot-column.

This is useful for moving sprites with the joystick. You can also use this to move a sprite off the screen if you want to hide it temporarily, using CALL LOCATE to send it to a dot-row value higher than 192.

CALL LOCATE needs the sprite #, a dot-row, and a dot-column. Her's how it would look:

CALL LOCATE (*#sprite number, dot-row, dot-column*)

CALL DELSPRITE. If you want to erase a particular sprite, you can use DELSPRITE to remove a sprite from the screen. All you have to do is specify the sprite you want deleted. The form for the command is:

CALL DELSPRITE (*#sprite number*)

If you want to remove all sprites from the screen, you could type:

CALL DELSPRITE (ALL)

Other Sprite Commands. The previous sprite commands commands are the ones you probably will want to use in your

games, and are the ones used in the game Moneybags. However, several other commands are used with sprites. They are:

- CALL COLOR, which can be used for sprites by specifying the sprite number (putting a # sign before it) and the number of the foreground color. You can use this to change colors of a sprite.
- CALL MOTION, which changes the motion of a sprite by specifying the sprite number, the row velocity, and the column velocity.
- CALL POSITION, which finds out where the sprite is by specifying the sprite number, and two variables, which contain the dot-row and dot-column. You could use this to see if a sprite has moved to a certain part of the screen.
- CALL DISTANCE, which tells you how far apart two sprites are by specifying the two sprite numbers and a variable that tells how far apart they are in terms of the *square* of the distance; you can also find out how far away a sprite is from a specific dot-row and dot-column, in terms of the square of the distance.
- CALL PATTERN, which changes a sprite to a new pattern by specifying the sprite number and the pattern shape in a way similar to the CALL CHAR command. You can use this to specify a pattern for a CALL MAGNIFY of 3 or 4, using one string of numbers rather than four different CALL CHAR commands.

Moneybags

Now that you've seen a short summary of how sprites work, you're ready to see sprites in action.

The game starts by drawing a dotted line on the screen to divide the highway. The title and score PRINT at the bottom.

A green money truck moves along the top of the screen. They must have left the back door open, because money is flying out of the door and scattering across the highway. As the truck speeds off, a bank guard jumps out of the truck. The guard's job is to get all of the money without being run over.

This seems easy enough, but cars come from both sides of the highway, six moving along the upper half of the highway and six more along the lower half. Each car is a different color and moves at a different speed.

13 Moneybags

You must guide the bank guard with your joystick, moving him next to a moneybag, and picking it up by pressing the fire button. Each moneybag you retrieve increases your score by 1.

But be careful. If one of the cars hits the bank guard, the game is over.

If the guard can pick up all the money on the screen, another truck will come along and more money will be scattered. This game requires not only fast action, but strategy. You have to figure out the cars' speeds so you won't be run over when you try to pick up the money.

Program 13-1. Moneybags

```
10 REM MONEYBAGS
20 CALL CLEAR
25 CALL SCREEN(16)
30 CALL CHAR(104,"0708107FFFFF1C1C")
31 CALL CHAR(106,"E09090FEFEFF3838")
32 CALL CHAR(105,"")
33 CALL CHAR(107,"")
35 CALL CHAR(112,"7F44447FFFFF1C1C")
36 CALL CHAR(114,"E09090FEFEFF3838")
37 CALL CHAR(113,"")
38 CALL CHAR(115,"")
40 CALL CHAR(120,"0709097F7FFF1C1C")
41 CALL CHAR(121,"")
42 CALL CHAR(122,"E01008FEFFFF3838")
43 CALL CHAR(123,"")
45 CALL CHAR(128,"0709097F7FFF1C1C")
46 CALL CHAR(129,"")
47 CALL CHAR(130,"FE2222FEFFFF3838")
48 CALL CHAR(131,"")
50 CALL MAGNIFY(3)
51 CALL CHAR(36,"1038103854102828")
52 CALL CHAR(37,"")
53 CALL CHAR(38,"")
54 CALL CHAR(39,"")
65 CALL CHAR(43,"F0F000000000000F")
66 CALL COLOR(2,10,1)
70 CALL HCHAR(12,1,43,32)
80 LET CR=1
81 LET CC=8*16-7
86 CALL CHAR(96,"7F7F7F7FFFFF1C1C")
87 CALL CHAR(97,"")
88 CALL CHAR(99,"")
89 CALL CHAR(98,"E09090FEFEFF3838")
90 LET P$="MONEYBAGS{5 SPACES}SCORE "
```

```
91 LET PLACE=2
92 GOSUB 1000
93 GOSUB 800
94 CALL SPRITE(#1,36,14,CR,CC,0,0)
95 LET COUNT=1 :: LET SCORE=0
96 GOSUB 200
97 LET P$=STR$(SCORE)
98 LET PLACE=23
99 GOSUB 1000
100 REM MAIN LOOP
122 FOR I=2 TO 13
123 CALL COINC(#1,#I,7,A)
125 IF A=-1 THEN 300
128 NEXT I
130 CALL JOYST(1,JC,JR)
133 IF JR=4 THEN 500
134 IF JC=4 THEN 550
135 IF JR=-4 THEN 600
136 IF JC=-4 THEN 650
150 CALL KEY(1,K,S)
155 IF K=18 THEN 900
190 GOTO 100
200 REM NEW SPRITES
201 RANDOMIZE
210 CALL SPRITE(#3,104,4,INT(RND*80+90)+1,1,
    0,RND*10+1)
212 CALL SPRITE(#12,104,3,INT(RND*80+90)+1,1
    ,0,RND*10+1)
220 CALL SPRITE(#4,112,12,INT(RND*80+90)+1,1
    ,0,RND*10+1)
222 CALL SPRITE(#13,112,5,INT(RND*80+90)+1,1
    ,0,RND*10+1)
230 CALL SPRITE(#5,104,14,INT(RND*80+90)+1,1
    ,0,RND*10+1)
240 CALL SPRITE(#2,112,10,INT(RND*80+90)+1,1
    ,0,RND*10+1)
250 CALL SPRITE(#6,120,6,INT(RND*72)+9,31*8,
    0,-RND*10-1)
252 CALL SPRITE(#10,120,11,INT(RND*72)+9,31*
    8,0,-RND*10-1)
260 CALL SPRITE(#7,120,7,INT(RND*72)+9,31*8,
    0,-RND*10-1)
262 CALL SPRITE(#11,128,13,INT(RND*72)+9,31*
    8,0,-RND*10-1)
270 CALL SPRITE(#8,128,8,INT(RND*72)+9,31*8,
    0,-RND*10-1)
280 CALL SPRITE(#9,128,9,INT(RND*72)+9,31*8,
    0,-RND*10-1)
285 RETURN
```

13 Moneybags

```
300 REM COLLISION
310 CALL SOUND(800,-5,0,880,0,890,0)
315 CALL CHAR(36,"18183C5A5A181824")
320 FOR I=1 TO 300
330 NEXT I
350 PRINT "YOU WERE RUN OVER BY A CAR"
360 GOTO 700
500 REM MOVE UP
510 IF JC<>0 THEN 150
520 IF CR>9 THEN LET CR=CR-8
530 CALL LOCATE(#1,CR,CC)
540 GOTO 150
550 REM MOVE RIGHT
560 IF JR<>0 THEN 150
570 IF CC<243 THEN LET CC=CC+8
580 CALL LOCATE(#1,CR,CC)
590 GOTO 150
600 REM MOVE DOWN
610 IF JC<>0 THEN 150
620 IF CR<175 THEN LET CR=CR+8
630 CALL LOCATE(#1,CR,CC)
640 GOTO 150
650 REM MOVE LEFT
660 IF JR<>0 THEN 150
670 IF CC>9 THEN LET CC=CC-8
680 CALL LOCATE(#1,CR,CC)
690 GOTO 150
700 REM ENDING
710 PRINT "YOU GOT ";SCORE;"BAGS OF MONEY"
720 PRINT "WANT TO PLAY AGAIN"
730 PRINT "PRESS Y FOR YES"
740 CALL KEY(3,K,S)
750 IF S=0 THEN 740
752 CALL DELSPRITE(ALL)
755 IF K=89 THEN 10
760 END
800 REM PRINT OUT MONEY
802 CALL SPRITE(#20,96,4,1,1,0,9)
804 LET COUNT=1
805 RANDOMIZE
810 CALL CHAR(136,"183C3C0000000000")
815 CALL COLOR(14,11,1)
820 FOR I=2 TO 31
830 LET BAG=INT(RND*22)+2
835 CALL GCHAR(BAG,I,Q)
837 IF Q<>32 THEN 830
840 CALL VCHAR(BAG,I,136)
846 NEXT I
848 CALL DELSPRITE(#20)
```

```
850 RETURN
900 REM GET MONEY
910 LET ROW=INT(CR/8)+1
915 LET COL=INT(CC/8)+1
920 CALL GCHAR(ROW,COL,M)
922 IF M<>136 THEN 100
925 LET SCORE=SCORE+1
927 LET COUNT=COUNT+1
928 IF COUNT>28 THEN GOSUB 800
930 CALL SOUND(100,-3,0,440,0)
932 LET P$=STR$(SCORE)
933 GOSUB 1000
950 CALL VCHAR(ROW,COL,32)
955 GOTO 100
1000 REM PRINT
1010 FOR I=1 TO LEN(P$)
1020 CALL VCHAR(24,I+PLACE,ASC(SEG$(P$,I,1))
     )
1030 NEXT I
1040 RETURN
```

Program Structure

Line 10 is the title, line 20 clears the screen, and line 25 colors the screen white.

Four different car shapes are created in lines 30-48. Lines 30-33 create one right-facing car; lines 35-38, a right-facing station wagon; lines 40-43, a left-facing car; and lines 45-48, a left-facing station wagon.

Line 50 is a CALL MAGNIFY (3) which makes each sprite normal sized, but of four characters. For instance, characters 104-107 make up one car, with the second and fourth characters blank, since the cars are 16 dots long and only 8 wide.

Lines 51-54 create the characters that make up the bank guard sprite. Since the bank guard is only an 8 dot by 8 dot shape, the other three characters are left blank.

Line 65 creates the character used for the striped median line of the highway. Line 66 colors it red, and line 70 puts it on the screen with CALL HCHAR.

Lines 80-81 set up the initial row and column of the bank guard.

Lines 86-89 create the group of characters for the money truck sprite.

Lines 90-92 set the string variable P$ which PRINTs the title on the screen by setting the PLACE variable, which is the column, and the subroutine at line 1000.

Line 93 sends the program to line 800 to move the money truck across the screen, scattering money.

Line 94 creates sprite #1, the bank guard, using character 36, color 14, row and column CR and CC, and 0 row and column velocity.

Line 95 shows how Extended BASIC can combine two commands on one line. The first command sets COUNT at 1 (the COUNT will be used to see how much money is still on the screen); the second command sets SCORE to 0. The two are separated by a double colon (::).

Line 96 uses the GOSUB command to create the 12 cars, sprites #2-#13.

Lines 97-99 put the score on the screen, using the subroutine that starts in line 1000.

Lines 100-190 are the main loop.

In lines 122-128 is a FOR/NEXT loop that tests to see if any of the sprites have collided with the bank guard sprite (sprite #1). The variable I is used in CALL COINC to test whether any one of the 12 sprites (numbered #2-#13) collided with sprite #1. A tolerance of 7 is used so that if the sprites overlap at all, a collision is detected. The collision result is stored in the variable A. Line 125 sends the program to line 300 if A = −1, indicating there was a collision.

Lines 130-136 read the joystick using the method from earlier chapters. Lines 150-155 read the joystick fire button.

Line 190 ends the main loop.

Lines 210-240 set up the six sprites that move from left to right. Notice that even though there are six sprites, only two shapes are used: One begins with character 104, the other begins with shape 112. Six colors are used, however. Each of the sprites is assigned a random column velocity, so that each moves at a different speed. Also, each sprite has a different dot-row value, created with a random number, so that they start out at varying rows. All of these right-facing cars start out between dot-row 91 and dot-row 170. This moves the cars along the bottom part of the screen.

Lines 250-280 set up the six sprites that move from right to left. They are similar to the cars moving from left to right except that they start in dot-column 31*8. (It's often convenient to multiply by eight so that you don't have to actually graph out a 256 × 256 grid. 31*8 is 248, so the cars will always start at the rightmost part of the screen.) Also, these cars are in dot-

rows from 9 to 80, so that they move along the top half of the screen. The sprites are made from characters that start with numbers 120 and 128. Once all these sprites are set up and moving, they will keep moving until one of them hits the bank guard. If they move off the edge of the screen, they'll just enter on the other side.

Lines 300-360 take care of the results if one of the cars collides with the bank guard. A sound is made in line 310, and the character of the bank guard is changed in line 315 so that the bank guard looks flattened. A delay is created in lines 320-330, and a message is displayed in line 350. The program then goes to line 700 where the ending routine executes.

Lines 500-690 move the bank guard, the up movement taking place in lines 500-540.

Line 510 checks to see if the joystick column variable is 0. If it isn't, there is a diagonal joystick movement, and the program returns to the main loop.

Line 520 checks to see if the bank guard's dot-row is greater than 9. If it is, it can move up. Line 520 shows the real power of TI Extended BASIC. TI BASIC only allows an IF/THEN to go to a line number, but Extended BASIC allows a statement to follow the THEN. In this case, IF CR is greater than 9, THEN 8 is subtracted from CR, by saying LET CR = CR − 8. This saves a separate line.

Line 530 uses CALL LOCATE to move the bank guard to the newly calculated row and column.

Line 540 sends the program back to the main loop.

Similarly, lines 550-590 move the guard right, lines 600-640 move down, and lines 650-690 move left.

Lines 700-760 end the game by printing the score and asking the player if another game is desired. Notice that line 752 was added. This deletes all the sprites with DELSPRITE (ALL). If this were omitted, the cars would still be on the screen when the game started over again.

The subroutine in lines 800-850 scatters the money on the highway.

Line 802 sets up the money truck sprite. It starts at the top left and goes to the top right. The speed is adjusted so it will reach the other side just as the last bit of money is put on the screen.

Line 804 sets COUNT back to 1, so that the variable can count to see if all the money has been picked up.

13 Moneybags

Lines 810-815 create the shape and color of the moneybag while lines 820-846 set up a FOR/NEXT loop which provides a random row for the moneybag to be placed in. Lines 835-837 test to see if something is already in the location, such as the median strip between the two halves of the road. If something *is* there, the program goes back and picks another random number in line 830. Lines 848 then deletes the money truck with CALL DELSPRITE.

Lines 900-955 operate the joystick fire button.

Lines 910-915 calculate the normal row and column from the dot-row and dot-column of the bank guard sprite, and line 920 uses CALL GCHAR to see if something is in the row and column.

Line 922 tests to see if the object CALL GCHAR found was character 136, which is the bag of money. If it didn't find the money, the program goes back to the main loop. If it did, line 925 adds 1 to SCORE.

Line 927 adds 1 to COUNT, so that line 928 can test to see if COUNT is greater than 28. If it is greater, there's only one bag left on the screen, so the subroutine in line 800 is called, which calls another money truck to scatter more money.

Line 930 creates the sound of picking up the money.

Lines 932-933 put the new score on the screen by using the subroutine at line 1000.

Line 950 puts a space where the money was. Notice that this has no effect on the bank guard sprite, because the sprites do not interact with screen characters and in fact are always put on top of any screen characters.

Line 955 returns this part of the main loop.

The subroutine in lines 1000-1040 puts a message on the screen. It uses P$ for the message, and the variable PLACE for the column.

Now that you've seen how this program works, you'll notice how simple the structure of this program is. Imagine trying to create a game that uses CALL VCHAR to move 13 objects around on the screen, and how slow it would be. This is the power of sprites.

Variables

CR	Bank guard row.
CC	Bank guard column.
P$	String variable used for screen messages.

PLACE	Variable used to put messages at a specific column.
COUNT	Determines when there is only one moneybag left on the screen.
SCORE	Number of moneybags recovered.
A	Variable used by CALL COINC to tell whether a collision occurred.
JC	Joystick column.
JR	Joystick row.
K	Key pressed.
S	Key Status.
BAG	Row in which a moneybag is put.
ROW	Variable for the normal row that a dot-row is translated into.
COL	Variable for the normal column a dot-column is translated into.
M	Variable used by CALL GCHAR to see if a moneybag was picked up.

Changing the Game

Now that you have seen how efficient sprites are, you may want to go back and rewrite some of the games that you saw earlier in this book.

- To make this game harder, you could occasionally test to see if a sprite is near the edge it did not start from. If it is, and you find it with a CALL POSITION, you could erase it and put it at a new dot-row so that the cars wouldn't always be in the same dot-row.
- You could scatter different values of money (shown by different colors) and get more points for picking up different moneybags.
- You could have some of the cars change lanes to get closer to the row the bank guard is in.
- You could even give the bank guard a time limit in which to get all the money.
- To make the game look different, change CALL MAGNIFY to 4, so that the cars and bank guard are large. Try this to see what it looks like; it may be too hard to play because there is no place for the bank guard to hide.

14 How to Create Your Own Game

14 How to Create Your Own Game

A game is, quite simply, a competition that takes place by following certain rules to achieve a desirable goal. All you need to create your own game ideas are three things: a goal, a competition, and rules. Add the programming building blocks that you have seen in earlier chapters, and you're ready to create original arcade-style videogames.

This chapter will give you a step-by-step process that will show you how to assemble all of the necessary pieces to make a complete game. The first five steps show you how to create your original game idea. The last ten steps show you how to make that game come alive by creating a computer program.

Step 1 is the most important, because it gives you a method to determine what the game's goal should be. What is the prize? What do you want to conquer, surpass, win, or beat? It's a good idea to plan beforehand, so that you won't program half of a game and suddenly realize it's not what you wanted. If you make plans and follow them through, you won't limit yourself by what you think the machine can do. Many beginning programmers limit themselves because they don't plan ahead.

Step 1: The Goal. In choosing a goal for your game, you can look at other types of games for ideas. For example, what is the goal of tennis, football, or chess? But you don't need to stop at ordinary games. Examine the goals of other pursuits, such as business, medicine, or politics.

Make a list of the goals that the people in these different professions strive for. Things you might include in your list would be: return the ball, make a touchdown, capture their king, deliver the products, serve the customers, remove the tumor, cure cancer, feed the poor, or end inflation.

Jot down as many goals as you can think of. When you have your list completed, exchange some of the words and see what you get. For example, instead of capturing the king, how about returning the king, or curing the king?

Another way to do this would be to make two lists. The first could be a list of types of actions, or verbs. The second list could include types of objects, or nouns. If your list includes verbs like eat, smash, or attack, and nouns like Martians, invaders, or potatoes, you could create goal-type ideas like smash the invaders, eat the potatoes, or attack the Martians. You could even program your computer to store these lists and pick combinations for you automatically.

Once you have a goal for your game, all you have to do is create a way to reach that goal. What do you smash the invaders with?

What eats up the potatoes? Or where do you attack the Martians? The answers to these questions create the competition of your game.

Step 2: Hero and Enemy. You need to have someone or something that is competing with the player for the same goal. Usually there is a hero (the player) and an enemy (the computer or another player). You must decide what the hero should be and what the enemy should be.

Usually the player will be a human character, but not always. The hero or the enemy can be Martians, mosquitos, or even potatoes.

Step 3: Location. The hero and villain competing for the same goal must have a location where they do this. Make a list of location possibilities, such as on the moon, in your backyard, or underwater. Make a location choice from this list.

Step 4: Weapons. Now that you know who is competing for what and where, all you need for the action to begin is an object the competing players can use. The object could be called a weapon.

The object is what the hero or villain uses to reach the goal. It's the basis for the nature of the competition of the game. In other words, it's what's used to smash, attack, eat with, or invade. Some weapons you might use could be a ball, a racquet, a gun, a boomerang, a book, an apple, a tank, a spaceship, or a laser beam.

Again, make a list of possibilities and from that list choose your weapons.

If you save all of these lists, you'll have a valuable resource for game ideas.

Step 5: Rules. Once you've chosen the goal and competition elements of the game, you need to formulate the rules of

how these things interact. The rules you choose will depend on what the goal is, where the location is, what the weapons are, and who the hero and enemy are.

It's important to determine what it takes to win or lose the game. How many points do you need? How do you get points? How do you lose points? How much time do you have? What happens if you go out of the boundaries? What kinds of moves can or can't be made? Think carefully about the other rules you might need. Ask yourself questions like these, and the answers will become the rules of your game. Write down the rules.

Now that your game has its rules, goals, the hero and enemy, the location, and weapons, you're ready to begin programming.

Step 6: Outline. The first step in creating the computer program for your game idea is to make a general outline on paper of what you want the program to do.

Review the building blocks presented in Chapters 2-6 if necessary. Or examine the games in Chapters 7-13 to see how different kinds of movement are achieved. Make notes on how your location background should be set up, how the hero should move, how the enemy should move, and how the weapons should work. You must consider how every piece of your program will have to work.

After examining all the pieces of the game idea, you can begin to create the actual computer program.

Now is a good time to get out graph paper and draw the various characters you'll want to use. Also, think about what kinds of colors you want to use in the design of your characters, what colors you want the screen to be, and what kinds of shapes and colors you might want to use for background objects. When you are thinking about this, write down what kind of sounds would enhance the game, and create a library of sounds that you like or that signify something, such as explosions, flying saucer sounds, or laser zaps.

Also, you should decide whether you want to use TI BASIC or TI Extended BASIC.

Now that you've planned the main elements of your game, the first and most important part of any game program is the main loop.

Step 7: The Main Loop. The main loop of a program usually does two things.

The first is that it reads the keyboard or joystick, so that it can take action based on player input. If the player does not do something, there is no real game. The main loop waits for player input.

The second thing that the main loop does is to keep track of time. After a certain number of times around the loop, various things can happen. You can use this timing to move the background, the enemy, or take other actions that make the game work.

The major parts of a game are decided in the main loop, so you should know what you want done here. Make it simple, because complicated things take time, and you want as little to actually happen in the loop as possible, so that the game plays as quickly as possible. BASIC is often slow enough without burdening the main loop with unnecessary calculations.

To keep the main loop from slowing down the game, you can make the more complicated things happen in smaller programs and subroutines, which the main loop goes to. Once a key is pressed, the joystick is moved, or the program has gone around the loop a certain number of times, the program can go to a smaller program or subroutine.

Write down what you want the main loop to do. Study main loops that have been presented in this book and write down the BASIC statements that you want to use to make this loop happen. Don't start using line numbers yet and don't type anything into the computer. You must patiently plan before you begin to actually program, to make sure that the program pieces work together.

Step 8: Smaller Programs and Subroutines. The smaller programs are accessed from the main loop, or sometimes from other smaller programs. Each smaller program or subroutine should do one or two things. When the smaller program is finished, it goes back to the main loop.

Smaller programs can do many things, but you want to keep them simple. If a smaller program is getting too complicated, you might want to break it down into several even smaller programs. The idea is to use smaller programs to break the game down into simple steps.

For example, one smaller program might move a character left, and another smaller program might move the same character right. A smaller program could fire a gun, move the background, or move the enemy. You can use smaller programs to

check to see if there will be a collision, to print or erase characters, to keep score, or keep track of time.

Often you can use a smaller program or subroutine in more than one game, because it will do the same thing. Keep a notebook and write down smaller programs that you might want to use from one game to another.

Write down the BASIC statements for each of your smaller programs or subroutines.

Step 9: The Line-by-Line Listing. Now that you've written down the BASIC statements you want to use for your main loop and your smaller programs, you must go through and see what needs to be done to make it a program.

Almost always you'll need an initial setup of things that must happen before the program goes to the main loop.

Part of this setup involves setting variables using LET statements. Write down all the variables that will be used in the program. Decide what their beginning values must be.

When this is done, write down the LET statements that make the variables what they should start out to be. Write down line numbers and have them be between 10 and 99, in the order that seems logical.

Next, think about all the beginning setups that your program may need, such as a background or putting your characters on the screen before you start. If you have room, you may want to print the instructions on the screen before you play the game.

You should also clear the screen, put a REM title in, color the screen, define characters with CALL CHAR, and do other things that need to be done before the game starts.

The game doesn't actually begin playing until the program gets to the main loop. Decide what you want to have happen before then, and put all the setups after the LET statements, but still between 10 and 99.

Now you're ready to write down exactly what you want in the main loop. Make sure that it loops back on itself with a GOTO statement at the end. Usually the main loop can use line numbers between 100 and 190.

Finally, one at a time, write down the smaller programs that you want to use, numbering each smaller program or subroutine starting at a multiple of 100. In other words, the first small program or subroutine should start at 300, the next at 400, and so on.

14 How to Create Your Own Game

Go over everything you've written. Now that you have a line-by-line listing of your program, you can begin to type in the actual statements.

Step 10: Typing the Setup and the Main Loop. Type in the LET statements that give your variables their beginning values. Then type in whatever else you want to happen before the main loop begins, such as the printing of the background, the beginning characters, the instructions, and colors.

Now, run the program and see if the background, characters, instructions, and other things are as you want them. It is easier to change them now, because whatever you do now will affect the whole program. Once you see the opening screen, you may want to make changes. If so, make your changes and note them.

After you have the opening the way you want, type in the statements for the main loop. To test the main loop, type in dummy statements that are the line numbers of each subroutine. Each dummy statement could be something like *XXX* PRINT "STOP IN LINE *n*" (*XXX* and *n* are the line numbers) and follow it with a STOP statement.

Then, run the program and test your mail loop. Press each key or move the joystick and see what happens. The program should go to the line you want it to, then stop. You can tell which line it went to by the "STOP IN LINE *n*" report.

If part of your main loop uses a counter which counts to a certain number before it jumps, test it by using PRINT to see what value the variable had when the program stopped.

When you're satisfied that your main loop works the way you want, you should SAVE the program on tape or disk. Always get in the habit of saving parts of your program as you write them. That way, when you make a mistake or accidentally erase part of the program, you can go back to the version you saved and continue working again from there.

Step 11: Typing the Smaller Programs and Subroutines. Now you're ready to type in each smaller program or subroutine. Type one in at a time and test it. Run the program and see if it does what it's supposed to do.

Always start with the simplest smaller program or subroutine and work up to the hardest. For example, if you have a smaller program to move the character, type that in first and test it. When you press the left-arrow key, see if the character moves to the left. Test the smaller programs thoroughly,

making sure that they work properly.

As each smaller program works the way you want it to, save the program. As you do more difficult smaller programs, test the old ones to make sure they still work. If you have trouble with your program at this point, read Appendix C, "In Case of Error," which contains helpful hints on how to detect errors.

Experiment with each smaller program, and see what happens when you make changes. Often you'll get a new idea as you are programming, and this is the perfect time to try it out. Because BASIC often tells you if you have an error, you can try out different statements and ideas, and see what effect they have on the program.

Step 12: Keep Records. Make notes on each experiment you make as you program. If you discover that something doesn't work, write it down. If you find something that you like, write that down too. You could even keep a separate notebook for smaller programs that can be reused in other games.

SAVE to tape or disk all the versions of your program as you create them. If you're using tape to store your program, write down in a notebook the counter number on your recorder which marks each place where you have a program version. Put the date on each version, so you can later remember what you were doing and be able to sort out each version.

Step 13: Test. When all the smaller programs have been added and tested, you can test the complete program game. Make sure that all the parts do what you want them to, and test to see that none of the characters can go off the screen, or that no characters pass through each other without something happening (unless you want it that way).

One good way to test your game at this point is to give it to a friend to play. Another is to put the game away for a few days and then go back to it and play the game as if you have never seen it before.

Think about each part of the game: How do the hero and villain interact with each other? How do they interact with the boundaries? Do the weapons do what they should? Does the scoring work? Are the characters moving as fast or as slow as you want them to?

Step 14: Corrections. Make any corrections or additions

necessary. Retest again and again. If you want your game to really work, you must test it. Nothing is worse than having a game that seems to work, but doesn't when the player does something that the programmer didn't think of. Try to see your game from another person's point of view, and forget for a moment how the actual program works.

When you make your corrections, test all the other parts of the game as well. Sometimes one error can cover up another, and sometimes you think you are fixing one error when you're actually causing another.

Step 15: Copies. When you're satisfied with the completed game and there are no mistakes in it, make two copies on two separate tapes.

Now your videogame is complete. You can sell it, give it to your friends, or play it yourself.

However, you should be warned that once you finish your first game, you'll want to go right back and create another that is even better.

Copyrighting and Marketing

Once you've created your own game, you have many options. Assuming that you didn't copy your ideas from a commercial game, you should probably copyright your game. Although your game is actually protected under copyright law once you've written it, you can insure this by putting a copyright notice in your program, or writing to the Library of Congress in Washington, D.C., for the papers needed to copyright your program.

To put a copyright notice in your game, just type, near the first line:

```
101 REM COPYRIGHT (C) 1984 YOUR NAME
```

and you are protected by common law copyright.

Once you've protected your game, you might want to submit it to a magazine. Magazines will pay you for your game and often will let you resell it elsewhere. When you sell a game to a magazine, make sure you know what rights you're selling.

You can also try to sell your game to a company that sells and distributes other videogames. If you want to do this, write to them, describing your game in a most general way, and wait for a reply. Don't send in your game until they ask you to. Usually they'll send you a form to fill out and they'll sign a

letter saying that they won't steal your idea.

You also could try to sell your game yourself. Many magazines have low rates for small ads at the back of the magazine, and you can sell your games by mail. However, if you do, make sure you have plenty of copies on hand in case you get a lot of orders. The Post Office has a regulation that says that if you sell by mail, you must send out all orders or offer the customer a refund within 30 days.

Often games are fun to play, but they may not be as good as the best that are for sale, or you may not want to get this involved. There's probably a TI users group in your area, and you might want to set up a tape exchange so you can trade programs that you have written for programs that others have written.

Whatever you do with them, writing games is fun for its own sake. You can create games that play *exactly* the way you want them to play, and when you get tired of the way they play, you can change them or write new ones. The best part of it all is that once you have your hands on a computer, you don't have to spend money to create an infinite number of games, all the games you could ever want.

If you want hundreds of exciting, fast-action arcade games, the TI has a lifetime supply built right in. All you need is your imagination and programming skill!

Appendices

Appendix A

Characters: ASCII Code Numbers and Sets

ASCII Code #	*Character*
Set #1	
32	(space)
33	!
34	"
35	#
36	$
37	%
38	&
39	'
Set #2	
40	(
41	)
42	*
43	+
44	,
45	-
46	.
47	/
Set #3	
48	0
49	1
50	2
51	3
52	4
53	5
54	6
55	7
Set #4	
56	8
57	9
58	:
59	;
60	<
61	=
62	>
63	?

ASCII Code #	*Character*
Set #5	
64	@
65	A
66	B
67	C
68	D
69	E
70	F
71	G
Set #6	
72	H
73	I
74	J
75	K
76	L
77	M
78	N
79	O
Set #7	
80	P
81	Q
82	R
83	S
84	T
85	U
86	V
87	W
Set #8	
88	X
89	Y
90	Z
91	[
92	\
93	]
94	^
95	—

A Appendix

ASCII Code #	*Character*	*ASCII Code #*
Set #9		**Set #13***
96	`	128
97	A	129
98	B	130
99	C	131
100	D	132
101	E	133
102	F	134
103	G	135
Set #10		**Set #14***
104	H	136
105	I	137
106	J	138
107	K	139
108	L	140
109	M	141
110	N	142
111	O	143
Set #11		**Set #15***
112	P	144
113	Q	145
114	R	146
115	S	147
116	T	148
117	U	149
118	V	150
119	W	151
Set #12		**Set #16***
120	X	152
121	Y	153
122	Z	154
123	{	155
124	¦	156
125	}	157
126	~	158
127	DEL	159

*There are no standard characters for sets 13 through 16. This has no effect on your ability to define them and use them in CALL HCHAR and CALL VCHAR statements, but it is very difficult to use them in PRINT statements.

Appendix B

Color Values

Color	Number Value
Transparent	1
Black	2
Medium Green	3
Light Green	4
Dark Blue	5
Light Blue	6
Dark Red	7
Cyan	8
Medium Red	9
Light Red	10
Dark Yellow	11
Light Yellow	12
Dark Green	13
Magenta	14
Gray	15
White	16

C Appendix

In Case of Error

When you're programming, you'll occasionally make mistakes. The TI computer has the ability to check for three kinds of errors, and you'll see a fourth method that you can use yourself.

The TI manual that came with your computer has an appendix in the back called Error Messages that can help you with certain kinds of errors. The three kinds the TI can find are:

- Errors that you make while typing in a line. These will usually be things that make no sense at all. You must retype the line so that it makes sense in terms of BASIC keywords and syntax.
- Errors that are caught just before the program starts running. These usually have to do with variables, DIM statements, being out of memory, and so on.
- Errors that cause the program to stop after it has started to run, such as a GOTO to a line number that does not exist or a value that is illegal for a HCHAR.

Anytime the TI tells you that you have made an error, you can look up the error message in the manual's index on Error Messages. Go over your program carefully to locate the error.

Tracing Errors. The TI has a special command called TRACE. By simply typing TRACE before you run a program, you can see how the program flows. You will see a series of line numbers enclosed by ⟨⟩ symbols, such as ⟨300⟩ ⟨310⟩ ⟨350⟩.

You can follow the logic of your program and if it crashes, where it stopped.

TRACE gives you two ways of finding errors. You can see where your program stopped, if it stopped and you don't understand the error message. You can also use it to see the logical flow of your program, which may help you to notice that the program is not going to the lines it should, in the proper order.

Type UNTRACE to turn off the TRACE command, so that you can run your program normally.

Logical Errors

This is the fourth and hardest kind of error to catch. Your program runs, you do not get an error message, yet something is wrong.

Here are some things to try:

- Use TRACE to follow the program logic again. See if you typed what you thought you did. Check your program to make sure you did not accidentally erase a line.
- Check all your GOTO statements to make sure you're not accidentally sending your program into a loop that it can't get out of.
- Check your FOR/NEXT loops to make sure you have written them correctly. If you have one loop inside another, make sure that the one inside ends before the next begins, and that your FOR/NEXT loops use different variables.
- Make sure you don't confuse variables that seem alike. A$, A, A$(2), and A(3) are all different *kinds* of variables. The first is a string variable, the second is a numeric variable, the third is a dimensioned string variable array, and the fourth is a dimensioned numeric variable array.
- Check your logic carefully when you use operatives such as AND, OR, NOT, or < >. What you *think* a statement means may not be what it actually means.
- In complicated statements, use parentheses to sort out mathematical operations. Don't type 3*A + 2/B unless you are sure which will be performed first.

How to Find Errors

Here are a few ways to find errors.

- If you're not sure what value a variable has in a process, you can always use PRINT AT to print out the variable somewhere on the screen.

For example, if you are working out a complicated idea like the creation of the river in "Riverboat," you could print the values of A and B on the side of the screen by using a generalized screen print routine with CALL VCHAR. You could put A and B into strings and use SEG$ to put them on the screen, one character at a time, in a location that won't interfere with the game play.

Then, when the program runs, you can see the values of A and B. In "Mushrooms," for example, you could PRINT the values of the arrays R and C to make sure that they contained the correct characters for each step of the program. You can always take out the generalized screen print statements after you are sure what you are doing, but they are very valuable if

you want to follow a process. If your screen is full and you have a printer, you can always print out variables on the printer.

- You can break your program into steps. Then, use the command BREAK to stop your program at a certain point. All you have to do is type BREAK (*line number*) and the program will stop when it comes to that line number. Using BREAK, you can see just how far your program got before it stopped. Start at the beginning of your program and work down.

Also, once your program has stopped at the line you think it should, you can use PRINT as a command by itself (with no line number) to print out variables to see what value a variable had at the moment the program halted.

If your program runs to the point where it should, and all the variables are what you think they should be, change the BREAK line number and pick a new line number further in the program.

Keep BREAKing your program until you have figured out where the problem is. Once you've narrowed the area where the error could be, you'll be able to track down the problem eventually.

You can turn off BREAK by typing UNBREAK.

- Put in fixed numbers. If you are using a counter and it doesn't seem to work, try setting a variable equal to a specific number and see what happens. Choose specific numbers to see if they do what you think they should. For example, if you have a counter that is supposed to count to 3, set it equal to 3 the first time, and see if it does what it's supposed to do. Similarly, test out all possibilities of a number. If you want a number between 3 and 20, try 3 and then try 20. The idea is to make sure that all numbers are allowable. Often, if you are dealing in random numbers, some of the numbers may be all right and others may not, but because they are random, you might not know which ones are causing your program to have problems. If you take the highest and lowest, you can assume that those in between will work correctly.
- If all else fails, take a rest. Put your program away and look at it later. Also, if you can't figure out a problem, another person who is familiar with the TI may have an insight into what is wrong with your program.

Index

Creating Your Own Game

You'll quickly be creating your own action games on the TI-99/4A when you use this step-by-step guide to game programming. Here's everything you need to design and write video games, including:

- Creating your own custom game figures and characters
- Moving figures around the screen
- Using the keyboard or joystick to control movement
- Vertical and horizontal scrolling of the screen
- Coloring the game characters and background
- Animation
- Drawing game mazes
- Making sound effects with the TI
- Creating and moving sprites with TI Extended BASIC
- A step-by-step process for designing and writing your own games
- How to test and debug a game
- And eight complete games for you to type in and play

Best of all, you'll see several finished games and how each works. You'll be able to follow the programs line-by-line, seeing how programming techniques work in game situations. You'll learn all the necessary techniques to create a wide variety of effective, exciting video games.

Whether you're just beginning to program on the TI-99/4A or have been computing for years, you'll find *Creating Arcade Games on the TI-99/4A* a valuable resource, a book you'll turn to again and again as you write action-packed video games.

ISBN 0-942386-27-2